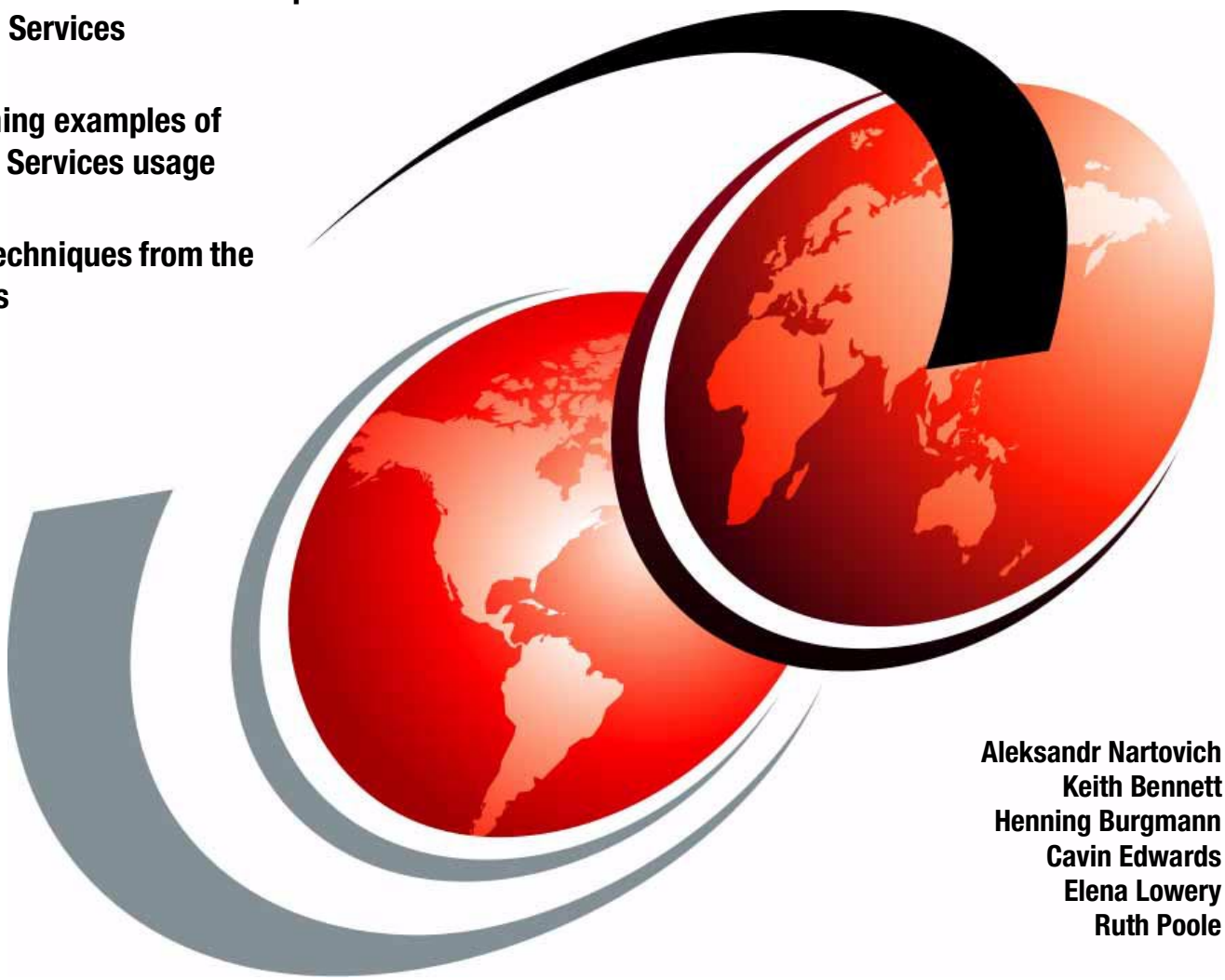
WebSphere® software

IBM

WebSphere Application Server Enterprise Edition 4.0

A Programmer's Guide

- The ultimate reference to WebSphere Enterprise Services
- Programming examples of Enterprise Services usage
- Tips and techniques from the developers



Aleksandr Nartovich
Keith Bennett
Henning Burgmann
Cavin Edwards
Elena Lowery
Ruth Poole

Redbooks

ibm.com/redbooks

International Technical Support Organization

WebSphere Application Server Enterprise Edition 4.0: A Programmer's Guide

February 2002

SG24-6504-00

Take Note! Before using this information and the product it supports, be sure to read the general information in “Special notices” on page vii.

First Edition (February 2002)

This edition applies to Version 4.0 of IBM WebSphere Application Server Enterprise Edition for use with the Windows NT and Windows 2000 operating systems.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JLU Building 107-2
3605 Highway 52N
Rochester, Minnesota 55901-7829

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2002. All rights reserved.
Note to U.S Government Users - Documentation related to restricted rights - Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Contents iii

Special notices vii

IBM trademarks viii

Preface ix
The team that wrote this redbook ix
Special notice xi
Comments welcome xi

Chapter 1. WebSphere Application Server Enterprise Edition overview 1
1.1 What is new in Version 4.0 2
1.2 WebSphere Enterprise Services 3

Chapter 2. Overview of the OrderEntry application 5
2.1 Introduction 6
2.1.1 The ABC Company 6
2.1.2 The ABC Company database 7
2.1.3 A customer transaction 7
2.1.4 Application flow 7
2.2 OrderEntry application database layout 10
2.2.1 Tables 10
2.3 Development environment 14

Chapter 3. The Business Rule Beans service 15
3.1 Overview 16
3.1.1 Benefits 16
3.1.2 BRBeans framework components 16
3.1.3 Identifying business rules 18
3.2 Implementing business rules with the BRBeans framework 18
3.2.1 Setting up the development environment 19
3.2.2 General approach for implementing business rules 22
3.2.3 Implementing a classifier rule 26
3.2.4 Implementing base rules 32
3.2.5 Implementing a business rule with dependent rules 41
3.3 Performance considerations 46
3.4 Deployment 47
3.5 Implementing the CustomerClassification class 48

Chapter 4. Extended Messaging Service 51
4.1 An overview of basic JMS and JMS Listener 52
4.2 Configuring JMS point-to-point for the OrderEntry application 53
4.2.1 The core components of JMS point-to-point 54
4.2.2 Defining physical queues 55
4.2.3 Defining JMS resources 55
4.2.4 WebSphere Application Server 60
4.2.5 JMS Listener XML configuration file 65
4.2.6 The message bean 67
4.2.7 Testing the JMS sample code 74

© Copyright IBM Corp. 2002

4.3 JMS publish/subscribe 76
4.3.1 The core components of JMS publish/subscribe 77
4.3.2 MQSeries message broker 77
4.3.3 The JMSAdmin tool 78
4.3.4 WebSphere Application Server 80
4.3.5 JMS Listener XML configuration file 81
4.3.6 The message bean for the publish/subscribe model 82
4.4 Advanced JMS facilities 84
4.4.1 Transaction support 84
4.4.2 JMS Listener threads and MaxSessions 86
4.4.3 Exception handling 86
4.4.4 Security 88
4.4.5 MQSeries clustering 88
4.5 The future of WebSphere messaging 89

Chapter 5. The WorkArea service 91
5.1 WorkArea overview 92
5.1.1 WorkArea scope 92
5.2 Using WorkArea in the example application 93
5.2.1 Binding to the WorkArea facility 93
5.2.2 Creating and populating WorkArea 94
5.2.3 Accessing WorkArea 95
5.2.4 Using WorkArea 95
5.2.5 Terminating WorkArea 96
5.3 Beyond the example 96
5.3.1 Using property modes to protect information 97
5.3.2 Nested WorkAreas 97
5.3.3 Changing the modes 101
5.3.4 Other methods defined by the UserWorkArea interface 101
5.3.5 Putting other objects in WorkArea 102
5.3.6 CORBA considerations 102
5.4 Configuring the WorkArea service in WebSphere Application Server 102
5.4.1 Ensuring the WorkArea service is enabled in the Administrative Console 102
5.4.2 Setting maximum send and receive size 103
5.4.3 Passing WorkArea through multiple WebSphere Application Servers 104

Chapter 6. The Internationalization service 105
6.1 The Internationalization service overview 106
6.1.1 Computers located in different locales 106
6.1.2 Computers located in different time zones 106
6.1.3 The Internationalization service solution 106
6.2 Internationalization context 107
6.2.1 The information in the internationalization context 107
6.2.2 Types of context 107
6.2.3 The Internationalization Service API 108
6.2.4 Internationalization context management 111
6.3 Using the Internationalization service in the example application 112
6.3.1 Binding to the Internationalization service 112
6.3.2 Retrieving and using the caller internationalization context 113
6.3.3 Retrieving and using the invocation internationalization context 114
6.4 Enabling or disabling the Internationalization service 114
6.4.1 Enabling internationalization context for an application server 114
6.4.2 Enabling internationalization context within EJB Java application clients 116

6.4.3 Configuring the programming environment . . . 116
6.5 Beyond the example application . . . 116
6.5.1 Setting invocation locale and time zone . . . 117
6.5.2 Tracing the Internationalization service function . . . 117

Chapter 7. The ActiveX bridge service . . . 121
7.1 Introduction . . . 122
7.1.1 Supported languages and platforms . . . 123
7.2 The Visual Basic OrderEntry client application . . . 124
7.2.1 Designing an ActiveX client program . . . 124
7.2.2 ActiveX client interface . . . 126
7.2.3 Installing the client application . . . 126
7.2.4 The client application components . . . 127
7.3 Initializing the JVM . . . 128
7.4 Using a J2EE client container . . . 129
7.4.1 Client container approach . . . 129
7.4.2 No client container approach . . . 129
7.4.3 Optimized container approach . . . 131
7.4.4 Components of the client side EAR file . . . 131
7.4.5 Launching a standard Java J2EE application in a client container . . . 131
7.4.6 Launching a client container in an ActiveX process . . . 131
7.5 Accessing Java objects, methods, and fields . . . 133
7.5.1 Primitive data type conversions . . . 133
7.5.2 Accessing a Java class . . . 136
7.5.3 Accessing Java objects . . . 136
7.5.4 Invoking Java methods . . . 137
7.5.5 Handling arrays . . . 140
7.5.6 Threads . . . 141
7.6 Error handling . . . 142
7.7 Good programming guidelines . . . 144
7.7.1 Visual Basic guidelines . . . 144
7.7.2 Active Server Pages guidelines . . . 145
7.7.3 Client container guidelines . . . 147
7.8 Modifying the EAR file for the ActiveX J2EE client container . . . 148
7.8.1 Setting up the Main class . . . 148
7.8.2 Creating the ‘dummy’ client JAR . . . 149
7.8.3 Adding the EJB references . . . 150
7.8.4 Optimizing the EAR file . . . 151

Chapter 8. The CORBA services . . . 153
8.1 Positioning of CORBA . . . 154
8.1.1 Role of CORBA in J2EE . . . 157
8.2 Overview of WebSphere Application Server Enterprise Edition CORBA support . . . 159
8.2.1 WebSphere Application Server support scenarios . . . 159
8.2.2 Interoperability with an EJB . . . 161
8.2.3 C++ CORBA software development kit (SDK) . . . 162
8.3 C++ CORBA clients for the OrderEntry application . . . 163
8.3.1 Configuring the environment . . . 163
8.3.2 IBM client of the Customer EJB . . . 164
8.3.3 VisiBroker client of the Customer EJB . . . 177
8.3.4 Problem determination . . . 185
8.4 Connecting OrderEntry application to C++ CORBA server . . . 187
8.4.1 Configuring the environment . . . 188

8.4.2 Connecting to an Orbix server 190
8.4.3 Problem determination 206

Chapter 9. The Business Process Beans service 209
9.1 BPBeans basics 210
9.2 BPBeans programming model 212
9.2.1 Task 212
9.2.2 Resource. 214
9.2.3 Context 215
9.2.4 Outcome decider. 215
9.3 Generic steps in BPBeans development. 216

Appendix A. Downloading and installing the OrderEntry application 221
Setup instructions. 222
Environment 222
Downloading the OrderEntry application ZIP file 222
Unzipping the file 222
Description of the directory structure 223
Setting up the databases 224
Setting up the OrderEntry application database 224
Setting up the BRBeans database 224
Configuring WebSphere Application Server. 224
Installing the OrderEntry application 226
Importing the BRBeans rules 228
Running the OrderEntry application 229
ActiveX application client-specific instructions 229

Appendix B. Additional material 231
Locating the Web material 231
Using the Web material 231
System requirements for downloading the Web material 231
How to use the Web material 232

Related publications 233
IBM Redbooks 233
Other resources 233
Referenced Web sites 233
How to get IBM Redbooks 234
IBM Redbooks collections. 234

Index 235

Special notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

© Copyright IBM Corp. 2002

IBM trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

e (logo)®
IBM ®
AIX®
Balance®
CICS®
DB2®
Encina®
IBM®
Informix™
MQSeries®
Perform™
Redbooks Logo
Redbooks™
S/390®
Sequent®
SP™
TXSeries™
VisualAge®
WebSphere®
Approach®
Notes®

Other company trademarks

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Preface

With the advent of Java 2 Platform, Enterprise Edition (J2EE) application servers, companies are beginning to adopt the programming model that offers new levels of portability, flexibility, reuse, and scalability. Complementing J2EE technology, the emerging set of Web services standards arms companies with even greater business flexibility to compete in a changing world.

The most sophisticated e-businesses, however, need to extend these industry standards to meet important enterprise requirements. In response to the needs of the most demanding customers, IBM delivers the new WebSphere Enterprise Services that are part of WebSphere Application Server Enterprise Edition Version 4.0.

This IBM Redbook discusses the new WebSphere Enterprise Services and shows the best practices for using the services in the applications. The target audience for this book is the I/T specialist and application developers. Some parts of the book, such as Business Rule Beans and Business Process Beans, will be of particular interest to business domain experts.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization (ITSO), Rochester Center.

Aleksandr Nartovich is a Senior I/T Specialist in the IBM ITSO, Rochester Center. He joined the ITSO in January 2001, after working as a developer in the IBM WebSphere Business Components (WSBC) organization. During the first part of his career, Aleksandr was a developer in AS/400 communications. Later, he shifted his focus to business components development on WebSphere. Aleksandr holds a degree in Computer Science from the University of Missouri, Kansas City, as well as a degree in Electrical Engineering from Minsk Radio Engineering Institute. You can reach Aleksandr at: alekn@us.ibm.com

Keith Bennett works as an IT Specialist in IBM Global Services, UK. He has worked for IBM for two years in developing e-business applications in the financial sector using the WebSphere platform. He holds a degree in Physics from Oxford University. You can reach Keith at: BENNETTK@uk.ibm.com

Henning Burgmann is an Advisory I/T Specialist with IBM Global Services in Cologne, Germany. He has several years of experience in large service projects in the telecommunications industries. His areas of expertise include object-oriented development with C++ and Java focussed on distributed applications using DCE and CORBA as middleware. He has one and a half years of experience in developing and managing applications for the WebSphere Application Server. You can contact Henning at: Henning.Burgmann@de.ibm.com

Cavin Edwards is a Solution Test Specialist from IBM UK, based in Hursley. He has nine years of experience in object oriented and client/server computing. He has worked in all areas of the software life cycle, using languages and tools such as C++, Java, and Delphi, within OS/390, UNIX, and PC environments. For the last two years, he has worked in the Hursley Solution Test Group, with predominantly Java and Web-based software for a variety of test and customer-focused solutions. His latest role includes Team Leading Solution Test EJB Interoperability Test and Performance Analysis. You can contact him via e-mail at: cav_edwards@uk.ibm.com

© Copyright IBM Corp. 2002

Elena Lowery is a Software Engineer at IBM Rochester. She currently works in the @server Custom Technology Center where she designs and develops custom applications for IBM Customers and Business Partners. She is a Sun Certified Java Programmer with four years of experience in object oriented programming and Java. Her areas of expertise include Java, WebSphere, XML, and Internet software development. Elena can be contacted at: elowery@us.ibm.com

Ruth Poole is a Software engineer with the iSeries Custom Technology Center. She worked for several years writing code for the iSeries operating system. She is a Sun Certified Java Developer and currently assists iSeries customers in developing and deploying applications. You can contact Ruth via e-mail at: rjpoole@us.ibm.com

Many thanks to the following people for their contributions to this project:

Debasish Banerjee
Karri Carlson
Sharad Cocasse
Logan Colby
Eric Herness
Chris D Johnson
Frank Malin
Tom Musta
Jon K Peterson
Anh-khoa D Phan
Dianne Richards
Ray V Scott
Rohit Singh
Scott Waldner
Al White
David Zavala
IBM Rochester, Minnesota

Lou Degenaro
IBM Watson Research Center, NY

Phil Adams
John Alcorn
Mark Coats
Stephen Cocks
Donavon Johnson
Richard Mills
Galen Young
IBM Austin, TX

Krishna Byri
Hailong Mao
Heather Phung
Richard A. Sitze
Ray Stutzman
IBM Charlotte, NC

Kyoko Hamada
IBM Mountain View, CA

Tracy Gardner
Vernon Green
Catherine Griffin
Dave Vines
IBM Hursley, United Kingdom of Great Britain and Northern Ireland

Special notice

This publication is intended to help the developers to create Web applications for WebSphere Application Server Enterprise Edition using WebSphere Enterprise Services. The information in this publication is not intended as the complete specification of any programming interfaces that are provided by WebSphere Application Server Enterprise Edition Version 4.0. See the PUBLICATIONS section of the IBM Programming Announcement for WebSphere Application Server Enterprise Edition Version 4.0 for more information about what publications are considered to be product documentation.

Comments welcome

Your comments are important to us!

We want our IBM Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  `ibm.com/redbooks`

- Send your comments in an Internet note to:

  `redbook@us.ibm.com`

- Mail your comments to the address on page ii.

1

WebSphere Application Server Enterprise Edition overview

This chapter provides the high-level overview of WebSphere Application Server Enterprise Edition v4.0. It describes the present position of WebSphere Application Server Enterprise Edition in the market. The main interest here is the new part of the WebSphere Application Server Enterprise Edition called *WebSphere Enterprise Services*.

© Copyright IBM Corp. 2002

1.1 What is new in Version 4.0

WebSphere Application Server Enterprise Edition Version 4.0 is designed for the most e-business demanding group of enterprises. Its sophisticated software infrastructure enables the creation of a flexible distributed application. It offers leading-edge Web services and Java 2 Platform, Enterprise Edition (J2EE) runtime environment.

The most significant changes in Version 4.0 include:

- All three editions of WebSphere Application Server have the same code base. This means that customers can upgrade WebSphere Application Server more easily.
- WebSphere Application Server is Java 2 Platform, Enterprise Edition (J2EE) Version 1.2 certified. This means that it supports the Java standards that are part of the J2EE specification. However, IBM goes beyond the J2EE specifications and offers extended services (some of them are part of the new Version 1.3 of J2EE) in WebSphere Application Server Enterprise Edition.
- WebSphere Application Server Enterprise Edition consists of four major elements:
  - **WebSphere Application Server Advanced Edition Version 4.0**
  - **WebSphere Enterprise Services**: IBM services are aimed at sophisticated Java developers whose needs go beyond the current Web services and J2EE standards. By offering a collection of leading edge Enterprise Services that plug into and augment the core WebSphere Application Server Version 4.0, Advanced Edition product, IBM adds significant value to the base application server. These Enterprise Services provide customers the flexibility they need to quickly develop and adapt e-business applications to meet emerging business requirements.
  - **IBM TXSeries**: Supports the traditional procedural programming model. TXSeries is middleware used in a high-performance distributed transaction processing environment. When sheer high-speed transaction processing is the goal, TXSeries provides the solution in Distributed CICS, typically used for COBOL applications, or IBM Encina, typically used for C and C++ applications. TXSeries enables transaction distribution and coordination over a broad set of resource managers – including relational databases, IBM CICS/390 applications, and traditional record-oriented resources, such as VSAM datasets, all within a scalable, reliable, and secure context.
  - **IBM MQSeries**: Messaging enables application integration by allowing business applications to exchange information across different platforms through sending and receiving data as messages. MQSeries takes care of network interfaces, assures “once only” delivery of messages, deals with communications protocols, dynamically distributes workload across available resources, handles recovery after system problems, and helps make programs portable. Programmers can use their skills to handle key business requirements, instead of wrestling with underlying network complexities.
- WebSphere Application Server Enterprise Edition expands the capabilities of J2EE and Web services technologies along two important dimensions:
  - Enterprise-class functions
  - Enterprise-class quality of service (QoS)
- The array of platforms where WebSphere Application Server is available includes Windows NT/2000, AIX, and Sun Solaris.

As with previous releases of WebSphere Application Server Enterprise Edition, the new version offers a much broader range of functions compared to WebSphere Application Server Advanced Edition. Figure 1-1 presents this difference in a graphical manner.

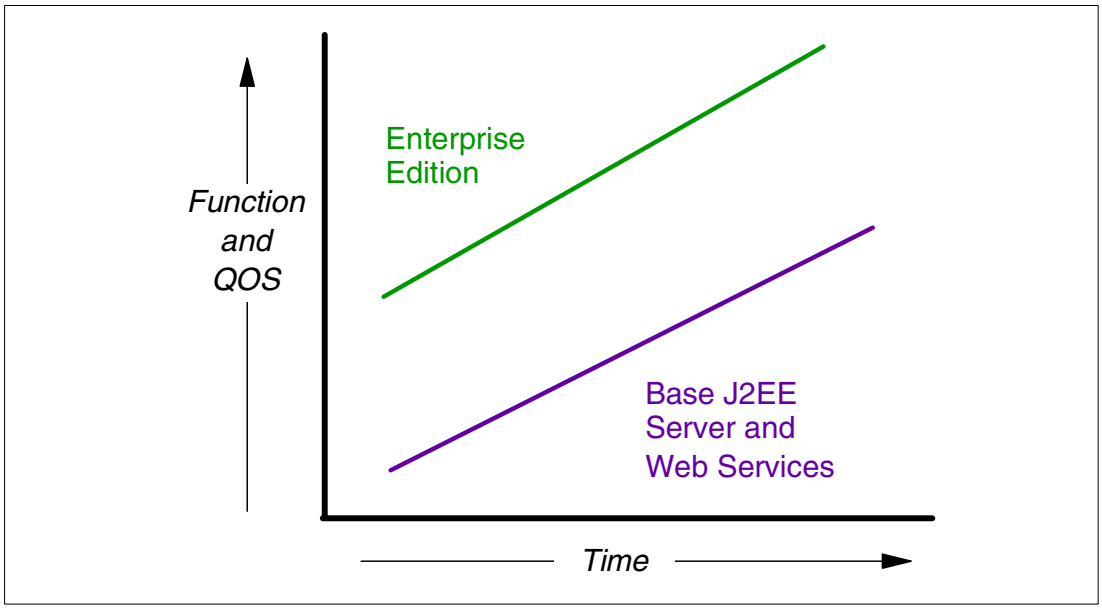


Figure 1-1 The difference of Enterprise Edition and Advanced Edition in time

As time goes by, WebSphere Application Server Enterprise Edition will still offer the additional functions and services on top of WebSphere Application Server Advanced Edition.

1.2 WebSphere Enterprise Services

The newest part of the WebSphere Application Server Enterprise Edition Version 4.0 is a group of eight services called WebSphere Enterprise Services:

- Business Rule Beans (BRBeans)
- Extended Messaging Service: Java Message Service (JMS) Listener
- Shared WorkArea
- Internationalization
- ActiveX bridge
- CORBA C++ Software Development Kit (SDK)
- CORBA interoperability
- Business Process Beans (BPBeans) as the technical preview

The primary goal that IBM tries to achieve with WebSphere Enterprise Services is to satisfy a growing demand from the enterprises to solve the most complex problems. As you can see from the list of WebSphere Enterprise Services, they include functionality that is not available in the J2EE v1.2 specification.

The IBM strategy is to expand WebSphere Application Server value by adding the new, enterprise-level services in two main areas:

- Enterprise-class functionality
- Enterprise-class quality of service (QoS)

WebSphere Enterprise Services in WebSphere Application Server Enterprise Edition Version 4.0 fall primarily into the category of enterprise-class functionality.

The second category of enterprise-class QoS is understood as enterprise-level scalability and performance. Even though, WebSphere Application Server Advanced Edition provides several QoS functions (the sophisticated Workload Management (WLM) algorithm, support for server clustering, improvements in the performance of Container Managed Persistence (CMP) entity beans, and the use of HTTP communication protocol between Web server and WebSphere Application Server), WebSphere Application Server Enterprise Edition targets more advanced functions and services for enterprise-class QoS.

All WebSphere Enterprise Services can be divided into three groups of functionality:

- **Integration and adaptability**: This group includes JMS Listener, CORBA interoperability, CORBA C++ SDK, ActiveX bridge, BRBeans, and WorkArea services.
- **Business process management**: This group is presented by the BPBeans service.
- **Globalization**: Internationalization service falls into this category.

The main support in WebSphere Application Server Enterprise Edition is added in the area of integration and adaptability. That's because the integration of the disparate application within a company is the most critical problem that the majority of enterprises face these days.

With J2EE Connector Architecture (included in WebSphere Application Server Advanced Edition) and WebSphere Enterprise Services, the enterprises get the runtime platform with the widest selection of connectors and integration services available on the market today.

Internationalization service is beneficial to the companies operating in the global environment. WebSphere Application Server Enterprise Edition sends (transparently to an application) the locale and time zone information from the client to the application server.

Finally, the business process management capabilities of BPBeans give the enterprises a powerful tool to:

- Overcome the problems with frequent changes to an application
- Provide a "rollback" capability at the business-process level
- Create an outcome-based application

Each of the WebSphere Enterprise Services is covered in more detail in the following chapters:

- Chapter 3, "The Business Rule Beans service" on page 15
- Chapter 4, "Extended Messaging Service" on page 51
- Chapter 5, "The WorkArea service" on page 91
- Chapter 6, "The Internationalization service" on page 105
- Chapter 7, "The ActiveX bridge service" on page 121
- Chapter 8, "The CORBA services" on page 153
- Chapter 9, "The Business Process Beans service" on page 209

2

Overview of the OrderEntry application

This chapter introduces the OrderEntry Web application that is representative of a commercial application. This application lies in the foundation of the programming examples we supply in this book. To make it easier to read this book, the OrderEntry application omits some of the attributes of the real-life application. This chapter also describes the layout of the database tables used in the application.

© Copyright IBM Corp. 2002

2.1 Introduction

The material in this redbook is covered based on the OrderEntry application. This application is used throughout the book to provide a hands-on experience with WebSphere Enterprise Services.

We show the advantages, offered by WebSphere Enterprise Services, in improving the application in several directions by:

- Integrating the OrderEntry application with other applications within the same fictitious company (CORBA services)
- Developing a new, message driven, entry channel for processing orders in the OrderEntry application (JMS Listener)
- Externalizing the business rules and making them easy to change without modifying the OrderEntry application
- Simplifying the code and making it less error-prone with the WorkArea service
- Adding the Internationalization service to process requests coming from different geographical regions
- Letting the existing ActiveX components communicate with the OrderEntry application by using the ActiveX bridge service

Since the BPBeans service is just a technical preview, we are not going to concentrate on the applicability of the service for the OrderEntry application. Rather, we highlight the benefits of the BPBeans service and outline the steps in developing a BPBeans-enabled application.

2.1.1 The ABC Company

The OrderEntry application is used by a fictitious company called ABC Company. The ABC Company is a wholesale supplier with one warehouse and 10 sales districts. Each district serves 3,000 customers (30,000 total customers for the company). The warehouse maintains stock for the 100,000 items sold by the Company. The ABC Company also operates a fulfillment center that is responsible for the shipment of the orders.

Figure 2-1 illustrates the company structure (warehouse, district, and customer).

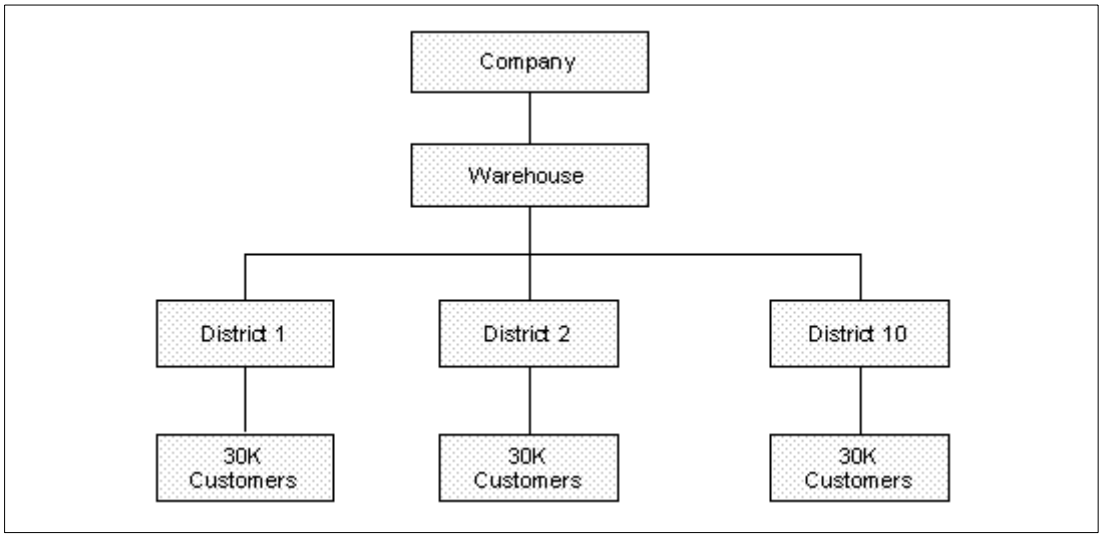


Figure 2-1 The company structure

2.1.2 The ABC Company database

The Company runs its business with a database. This database is used in a mission-critical, online transaction processing (OLTP) environment. The database includes tables with the following data:

- District information (next available order number, tax rate, and so on)
- Customer information (name, address, telephone number, and so on)
- Order information (date, time, shipper, and so on)
- Order line information (quantity, delivery date, and so on)
- Item information (name, price, item ID, and so on)
- Stock information (quantity in stock, warehouse ID, and so on)

2.1.3 A customer transaction

A customer transaction occurs based on the following series of events:

1. Customers telephone one of the 10 district centers to place an order.
2. The district customer service representative answers the telephone, obtains the following information, and enters it into the application:
   - Customer number
   - Item numbers of the items the customer wants to order
   - The quantity required for each item
3. The customer service representative may prompt for a list of customers or a list of parts.
4. The application then performs the following actions:
   a. Reads the customer last name, customer discount rate, and customer credit status from the Customer Table (CSTMR).
   b. Reads the District Table for the next available district order number. The next available district order number increases by one and is updated.
   c. Reads the item names, item prices, and item data for each item ordered by the customer from the Item Table (ITEM).
   d. Checks whether the quantity of ordered items is in stock by reading the quantity in the Stock Table (STOCK).
5. When the order is accepted, the following actions occur:
   a. A new row is inserted into the Order Table to reflect the creation of the new order (ORDERS).
   b. A new row is inserted into the Order Line Table for each item in the order.
   c. The quantity in the Stock table is reduced by the quantity ordered.
   d. A confirmation message is returned to the customer service representative
6. An employee at the fulfillment center accesses the Order table and ships the merchandise based on the incoming orders.

2.1.4 Application flow

To better understand the application's logic, we provide the OrderEntry application flow diagrams. The overall flow of the execution of a customer order request is divided into three diagrams:

- Getting customer information (Figure 2-2)
- Adding selected items to the shopping cart (Figure 2-3 on page 9)
- Checking out (Figure 2-4 on page 10)

The first diagram, getting customer information, is presented in Figure 2-2.

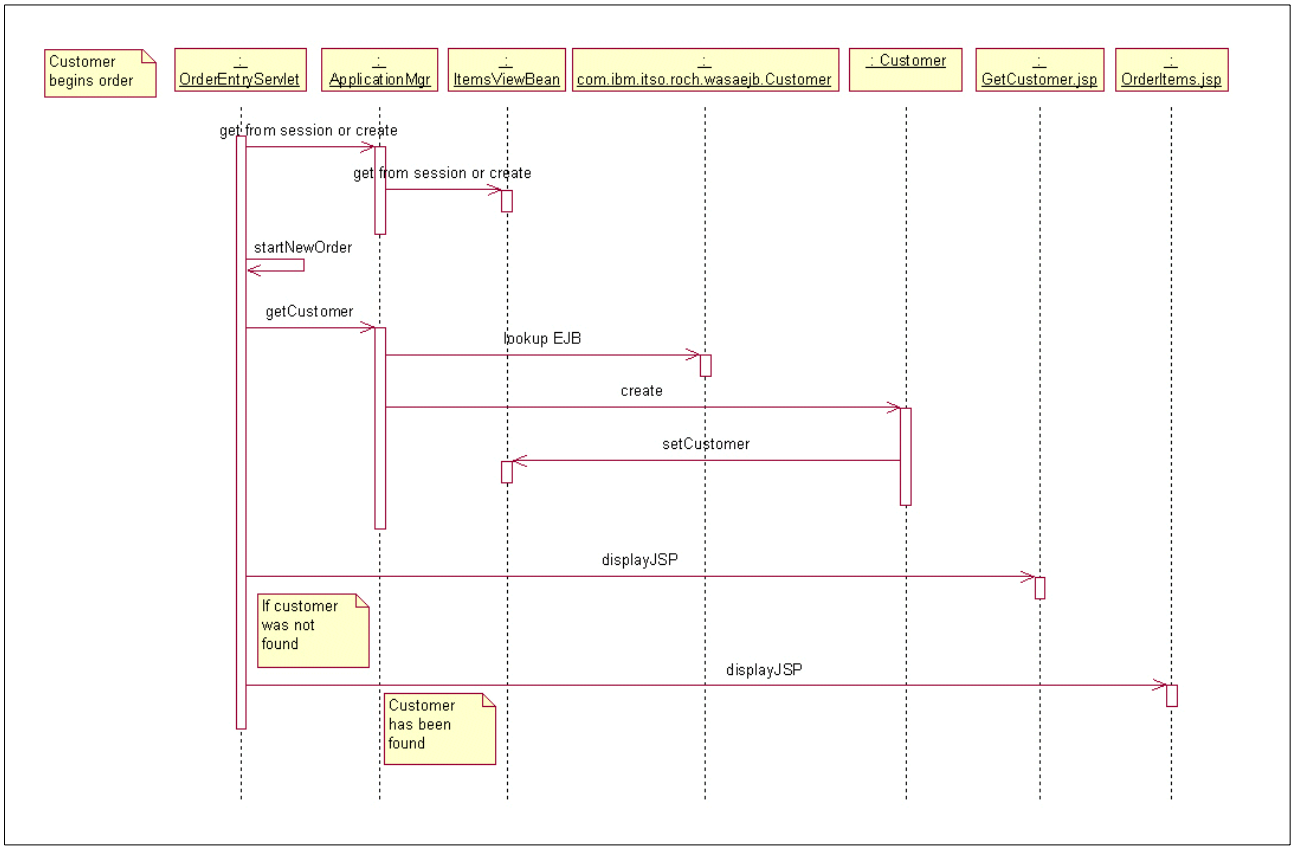


Figure 2-2 Getting customer information

A customer is searched by the customer ID number. If the customer is not found, the error message is displayed (OrderEntry application doesn't have a Web interface to create a customer).

If the customer is found, a new OrderItems page is displayed. This is the page where a customer service representative can add any number of items to the shopping cart. The process of adding the order items to the shopping cart is shown in Figure 2-3.

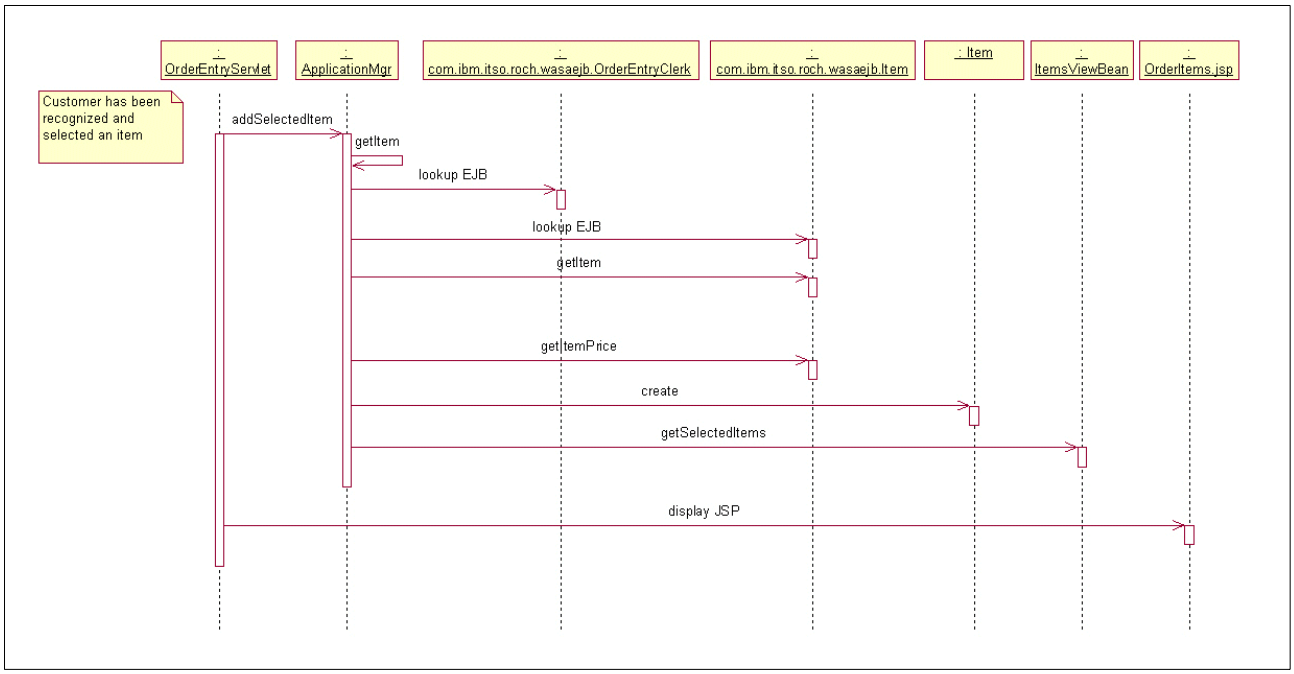


Figure 2-3 Adding items to the shopping cart

The application offers two ways to add items:

- By specifying an item ID and quantity
- By opening an item's catalog view and selecting items from the catalog

Only the first method is shown in the diagram.

When all items are selected, a customer service representative can check them out. This process is shown in Figure 2-4.

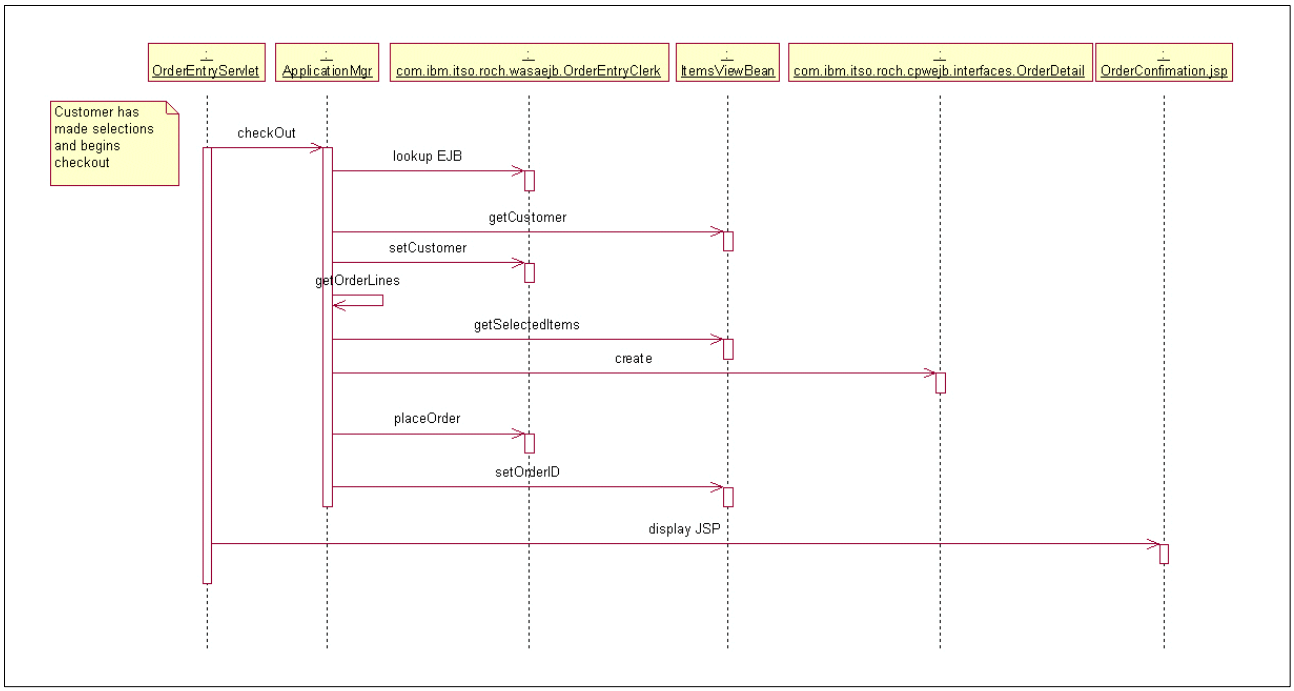


Figure 2-4 Order check out process

The checking out functionality is embedded into the *placeOrder()* method of the `com.ibm.itso.wasaejb.OrderPlacement` session EJB (not shown in the diagram). This method is called from the `com.ibm.itso.roch.wasaejb.OrderEntryClerk` session EJB.

2.2 OrderEntry application database layout

The sample application uses the following tables of the database:

- District
- Customer
- Order
- Order line
- Stock
- Item (catalog)

The following sections describe, in detail, the layout of the database.

2.2.1 Tables

The District table keeps information related to each district. The primary use of this table in the application is to store the next available order number. Table 2-1 shows the layout of the District table.

Table 2-1 District table layout (Dstrct)

Field name	Real name	Type	Length
DID	District ID	Decimal	3
DWID	Warehouse ID	Character	4
DNAME	District Name	Character	10

Field name	Real name	Type	Length
DADDR1	Address Line 1	Character	20
DADDR2	Address Line 2	Character	20
DCITY	City	Character	20
DSTATE	State	Character	2
DZIP	Zip Code	Character	10
DTAX	Tax	Decimal	5
DYTD	Year to Date Balance	Decimal	13
DNXTOR	Next Order Number	Decimal	9
Primary Key: DID and DWID			

The Customer table is used to store customer information. The search of the customer is done based on the customer ID column. To simplify the OrderEntry application, we decided not to implement additional customer search functions. Table 2-2 shows the Customer table layout.

Table 2-2 Customer table layout (CSTMR)

Field name	Real name	Type	Length
CID	Customer ID	Character	4
CDID	District ID	Decimal	3
CWID	Warehouse ID	Character	4
CFIRST	First Name	Character	16
CINIT	Middle Initials	Character	2
CLAST	Last Name	Character	16
CLDATE	Date of the last order	Decimal	8
CADDR1	Address Line 1	Character	20
CCREDT	Credit Status	Character	2
CADDR2	Address Line 2	Character	20
CDCT	Discount	Decimal	5
CCITY	City	Character	20
CSTATE	State	Character	2
CZIP	Zip Code	Character	10
CPHONE	Phone Number	Character	16
CBAL	Balance	Decimal	7
CCRDLM	Credit Limit	Decimal	7
CYTD	Year to Date	Decimal	13
CPAYCNT	Payment	Decimal	5

Field name	Real name	Type	Length
CDELCNT	Delivery Qty	Decimal	5
CLTIME	Time of Last Order	Numeric	6
CDATA	Customer Information	Character	500
Primary Key: CID, CDID, and CWID			

The Order table stores the order related information. The information in the Order ID column is used to key the order lines. Table 2-3 shows the Order table layout.

Table 2-3 Order table layout (ORDERS)

Field name	Real name	Type	Length
OWID	Warehouse ID	Character	4
ODID	District ID	Decimal	3
OCID	Customer ID	Character	4
OID	Order ID	Decimal	9
OENTDT	Order Date	Numeric	8
OENTTM	Order Time	Numeric	6
OCARID	Carrier Number	Character	2
OLINES	Number of Order Lines	Decimal	3
OLOCAL	Local	Decimal	1
Primary Key: OWID, ODID, and OID			

The Order Line table stores information about each item being ordered. The Order ID column keys each order line to the corresponding order. Table 2-4 shows the Order Line table layout.

Table 2-4 Order Line table layout (ORDLIN)

Field name	Real name	Type	Length
OLOID	Order ID	Decimal	9
OLDID	District ID	Decimal	3
OLWID	Warehouse ID	Character	4
OLNBR	Order Line Number	Decimal	3
OLSPWH	Supply Warehouse	Character	4
OLIID	Item ID	Character	6
OLQTY	Quantity Ordered	Numeric	3
OLAMNT	Amount	Numeric	7
OLDLVD	Delivery Date	Decimal	8
OLDLVT	Delivery Time	Decimal	6
OLDSTI	District Information	Character	24
Primary Key: OLWID, OLDID, OLOID, and OLNBR			

The Item table plays a role of the catalog in the OrderEntry application. Table 2-5 shows the layout of the Item table.

Table 2-5 Item table layout (ITEM)

Field name	Real name	Type	Length
IID	Item ID	Character	6
INAME	Item Name	Character	24
IPRICE	Price	Decimal	5
IDATA	Item Information	Character	50
Primary Key: IID			

The Stock table keeps the inventory of the items up-to-date. Table 2-6 shows the layout of the Stock table.

Table 2-6 Stock table layout (Stock)

Field name	Real name	Type	Length
STWID	Warehouse ID	Character	4
STIID	Item ID	Character	6
STQTY	Quantity in Stock	Decimal	5
STDI01	District Information	Character	24
STDI02	District Information	Character	24
STDI03	District Information	Character	24
STDI04	District Information	Character	24
STDI05	District Information	Character	24
STDI06	District Information	Character	24
STDI07	District Information	Character	24
STDI08	District Information	Character	24
STDI09	District Information	Character	24
STDI10	District Information	Character	24
STYTD	Year to Date	Decimal	9
STORDERS	Number of orders	Decimal	5
STREMORD	Number of remote orders	Decimal	5
STDATA	Item Information	Character	50
Primary Key: STWID and STIID			

2.3 Development environment

The OrderEntry application has been developed using the new tool called WebSphere Studio Application Developer. This tool integrates most of the capabilities available from other IBM tools such as WebSphere Studio and VisualAge for Java. In addition, WebSphere Studio Application Developer provides the GUI editors to specify the deployment descriptors according to the J2EE specification. We recommend that you use WebSphere Studio Application Developer as the IDE for the examples provided in this book.

For more information about WebSphere Studio Application Developer, refer to the Web page at: http://www.ibm.com/software/ad/studioappdev/

3

The Business Rule Beans service

This chapter explains how to use the Business Rule Beans service in building enterprise applications. It includes the following topics:

- Overview of the Business Rules Beans service
- Implementation of different business rules
- Performance considerations
- Deployment

Note: To understand the structure of the OrderEntry application and to run it, read the instructions in Appendix A, “Downloading and installing the OrderEntry application” on page 221.

© Copyright IBM Corp. 2002

3.1 Overview

The Business Rules Beans (BRBeans) service is a framework for externalizing implementation of business rules. The framework eliminates the need to redeploy application code after changing business logic. This is done by decoupling business rules implementation from the application code that uses them.

The following terms are used throughout this chapter:

- **Business rule**: A description of a business process or practice.
- **Business logic**: A part of program code that implements a business rule.
- **BRBeans rule**: A business rule implemented in the BRBeans framework.
- **Rule client**: Application code that uses a BRBeans rule.
- **Rule implementor**: A Java class that implements a business rule.
- **Trigger point**: A place in the rule client where a call to fire a BRBeans rule is made.

3.1.1 Benefits

The main benefits of BRBeans include:

- **Isolation of volatile business logic from application code**: Business logic and application code are implemented in different compilation units. The rule client can be implemented in Java classes, servlets, or Enterprise JavaBeans (EJBs). Business rules must be implemented as Java classes. The rule client knows the BRBeans rule name and interface, but it is not aware of the rule implementation.
- **Keeping knowledge of business rules external to the application**: Variable business rule data is not embedded in the rule client and can be easily accessed with the Rule Management Application.
- **Administrative maintenance of business logic**: Application behavior can be changed through an administrative process using the Rule Management Application. The tool can modify business logic in two ways:
  - By specifying a different rule implementor for a BRBeans rule name
  - By changing variable business rule data

3.1.2 BRBeans framework components

The BRBeans framework consists of EJBs, APIs, and stand-alone tools for maintaining business rules. The following components comprise the framework:

- BRBeans EJBs provide a runtime environment for the framework. During the development phase, they persist business rule configurations. At runtime, BRBeans EJBs are responsible for finding and firing BRBeans rules. BRBeans EJBs must be deployed with the application that uses them.
- BRBeans Trigger Point framework provides an interface to access BRBeans rules from the rule client. To use a BRBeans rule, the rule client creates an instance of a `TriggerPoint` (TP) object and calls one of the TP object's trigger methods (trigger, triggerClassifier, triggerSituational). Then TP object calls the *fire* method on the `RuleImplementor` object.
- BRBeans Rule Implementors are Java classes that provide implementation for common business logic, such as a number range check. These rule implementors are ready for use by rule clients through the BRBeans framework. BRBeans Rule implementors can be used as examples for implementing custom business rules.
- The Rule Management Application is a stand-alone tool for maintaining BRBeans rules. The tool allows you to explore and modify BRBeans rule configuration data: rule name, implementation class, start and end date, initialization parameters, and others.

- BRBeans Rule Management APIs can be used by developers to build custom rule management applications for use by domain experts.
- BRBeans Rule Importer and Exporter tool allows you to import and export BRBeans rule configurations as XML files. This tool is similar to WebSphere's XMLConfig tool; it can be used to move BRBeans rule configurations between servers.

The described components are used both during the development phase and at runtime. Three activities take place during the development process (see Figure 3-1):

- Rule implementors are created. This step is optional. Application developers can use BRBeans rule implementors provided with the framework.
- Trigger points are placed in the rule client.
- Business rules are configured in the Rule Management Application. The tool uses BRBeans EJBs to save BRBeans rule configuration information. This condition implies that an application server must be started prior to using the tool.

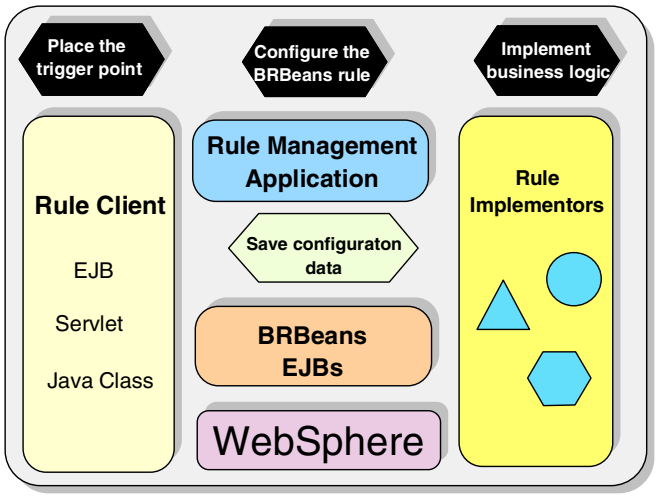


Figure 3-1 BRBeans components during the development phase

At runtime, the rule client invokes BRBeans rules using the `TriggerPoint` object. Each BRBeans rule configured in the Rule Management Application is represented as an EJB at runtime. After the BRBeans rule is located, the framework fires the actual implementation and returns results to the rule client. Figure 3-2 shows the interaction of BRBeans components.

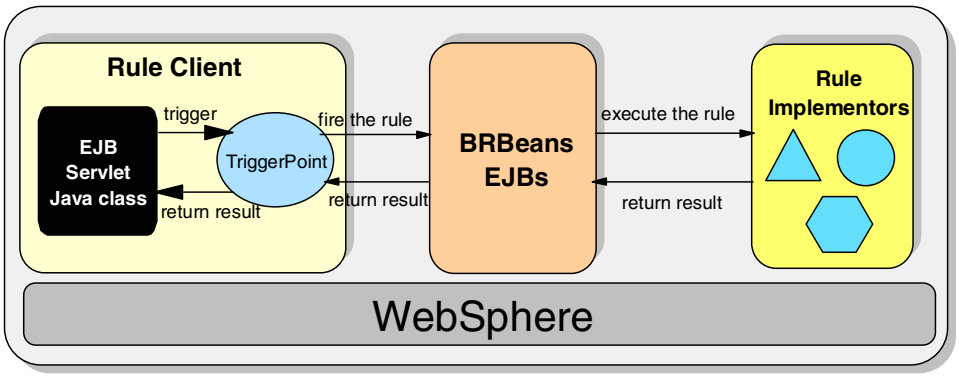


Figure 3-2 Runtime interaction of BRBeans components

3.1.3 Identifying business rules

Business rules are policies used by companies in conducting business practices. These policies may be specific to a company or may come from an outside regulatory agency. A business rule usually contains some variable information reflecting a change in a business process. For example, consider the following business rules:

- **Example 1**: If a customer has purchased $500 worth of products YTD, then the customer qualifies for the Preferred Club.

  This business rule is an example of an internal business policy. The balance required to qualify for the Preferred Club may change. In addition, the company may decide to add additional customer classifications (Gold Club and so on).

- **Example 2**: If the total order amount is greater than $100, then provide a 10 percent discount.

  This business rule is another example of an internal business practice. Variable information includes the total order amount and the discount rate.

- **Example 3**: The capital gains tax rate is 28 percent for stock sold during 18 months from the date of purchase and 20 percent for stock sold after 18 months from the date of purchase.

  This business rule is an example of an external government regulation. Tax rate and time periods are the variable data.

From the BRBeans framework perspective, all business rules belong in one of two categories:

- Classification rules
- Base rules

Classification rules return a classification based on factors involved in a business process. The BRBeans framework requires classification rules to return an object type of `java.lang.String`. A classification rule is demonstrated in Example 1. The rule implementor must return a string that represents a customer classification (Preferred, Gold) based on the customer's YTD purchases.

The rest of the business rules fall into the base rules category. In the BRBeans framework, base rules return an array of `java.lang.Objects`. Implementations of business rules in Example 2 and Example 3 above must return an array with one object of type `java.lang.Float` or `java.lang.Integer`.

Note: Primitives must be passed as `java.lang.Object` using primitive wrapper classes.

3.2 Implementing business rules with the BRBeans framework

This chapter shows how to implement one business rule using the BRBeans framework and examine implementations of several other business rules. We provide instructions for developing business rules in WebSphere Studio Application Developer. WebSphere Studio Application Developer tool provides an easy and convenient way to create rule implementors and package them in an Enterprise Archive (EAR) file for deployment in WebSphere Application Server.

If you don't have access to the tool, you can develop rule implementors in other Java editors and use WebSphere Application Assembly Tool to package the EAR file. We assume that DB2 is used to persist BRBeans rule data. The completed application is deployed in WebSphere Application Server Enterprise Edition.

The following sections explore the necessary steps to develop, deploy, and run the BRBeans enabled application.

3.2.1 Setting up the development environment

This section goes through the steps of setting up the environment for developing BRBeans. However, before you start executing the steps, make sure you have completed necessary prerequisites that are described in Appendix A, “Downloading and installing the OrderEntry application” on page 221. Then, complete the following steps:

1. Import OrderEntry application to WebSphere Studio Application Developer:
   a. Start WebSphere Studio Application Developer.
   b. Click **File-> Import**.
   c. Double-click the **EAR file**.
   d. Browse to **%OE_HOME%\OrderEntryWithoutBRBeans.ear**.
   e. Type the name of the project, which is `OrderEntry`.
   f. Click **Finish**.
2. Add the BRBeans JAR file to the OrderEntry EAR project:

   Note: You can perform this step using the WebSphere Application Assembly tool.

   a. In WebSphere Studio Application Developer, select the **OrderEntry** project.
   b. Click **File-> Import**.
   c. Double-click the **EJB JAR file**.
   d. Browse to **%WAS_HOME%\Enterprise\BRBeans** and select **BRBeansDB2.jar**.
   e. Type the name of the EJB project, which is `BRBeans`.
   f. Click **Finish**.
   g. In the task window, you can see a number of errors caused by BRBeans missing references. We need to add two JAR files to BRBeans EJB project classpath:
      i. Open the J2EE Perspective; click **Perspective-> Open-> J2EE**.
      ii. Click the **Navigator** tab.
      iii. Right-click the **BRBeans** folder and select **Properties**.
      iv. In the Properties window, select **Java Build Path**.
      v. Click the **Libraries** tab.
      vi. Click the **Add External Jars** button.
      vii. Navigate to **%WAS_HOME%\lib**.
      viii. Select **brbClient.jar** and **brbServer.jar**.
      ix. Click **OK** to save the properties.
      x. In the Navigator view, right-click the **BRBeans** folder and select **Rebuild project**.
   h. Add BRBeans EJBs references to the OrderEntryWeb module:
      i. In WebSphere Studio Application Developer, open the J2EE Perspective; select **Perspective-> Open-> J2EE**.
      ii. Click the **J2EE View** tab.
      iii. Expand the **Web Modules** folder.
      iv. Double-click **OrderEntryWeb**. This opens the web.xml file in the Editor view.

v. Click the **References** tab.

vi. Add three EBJ references as shown in Table 3-1.

vii. Save the changes.

Table 3-1 EJB references for BRBeans

EJB reference	Type	Home	Remote
ejb/com/ibm/ws/brb/Rule	Entity	com.ibm.ws.brb.RuleHome	com.ibm.ws.brb.Rule
ejb/com/ibm/ws/brb/RuleFolder	Entity	com.ibm.ws.brb.RuleFolderHome	com.ibm.ws.brb.RuleFolder
ejb/com/ibm/ws/brb/RuleHelper	Session	com.ibm.ws.brb.RuleHelperHome	com.ibm.ws.brb.RuleHelper

i. Change the JNDI names of BRBeans rules EJBs to be specific for your application:

i. In WebSphere Studio Application Developer, open the J2EE Perspective; select **Perspective-> Open-> J2EE**.

ii. Select the **J2EE View** tab.

iii. Expand the **EJB Modules** folder.

iv. Right-click **BRBeansDB2** and select **Open With-> EJB Extension Editor**.

v. Select the **Bindings** tab.

vi. On the Bindings tab, expand **BRBeansDB2**.

vii. Expand each EJB and change JNDI names (for each bean and the beans it references), respectively, to:

```
brbeans/OrderEntry/Rule
brbeans/OrderEntry/RuleFolder
brbeans/OrderEntry/RuleHelper
```

Note: Make sure to expand each EJB and change JNDI names for each EJB reference.

viii. Save the changes.

ix. In the Navigator view, select the folder under **OrderEntryWeb-> webApplication-> WEB-INF**.

x. Double-click the **ibm-web-bnd.xmi** file.

xi. Select the **Source** tab.

xii. Change the JNDI names of BRBeans EJBs as shown in Example 3-1.

xiii. Save the changes.

Example 3-1 Modifying the ibm-web-bnd.xmi file

```
...
<ejbRefBindings xmi:id="EjbRefBinding_4" jndiName="brbeans/OrderEntry/Rule">
    <bindingEjbRef href="WEB-INF/web.xml#EjbRef_4"/>
</ejbRefBindings>
<ejbRefBindings xmi:id="EjbRefBinding_5" jndiName="brbeans/OrderEntry/RuleFolder">
    <bindingEjbRef href="WEB-INF/web.xml#EjbRef_5"/>
</ejbRefBindings>
<ejbRefBindings xmi:id="EjbRefBinding_6" jndiName="brbeans/OrderEntry/RuleHelper">
    <bindingEjbRef href="WEB-INF/web.xml#EjbRef_6"/>
</ejbRefBindings>
...
```

j. Generate the BRBeans code for deployment:

i. In WebSphere Studio Application Developer, open the J2EE Perspective; click **Perspective-> Open-> J2EE**.

ii. Expand the **EJB Modules** folder.

iii. Right-click **BRBeansDB2** and select **Generate-> Deploy and RMIC Code**. Make sure three EJBs are selected: **Rule**, **RuleFolder**, and **Rule Helper**.

iv. Click **Finish**.

Note: For details on importing the BRBeans.jar file, including the installation for different database types (Oracle, SQLServer, Sybase, and Informix), see the BRBeans product documentation at:
`http://www.ibm.com/software/webservers/appserv/doc/v40/aee/index.html`

3. Add several JAR files to OrderEntryWeb classpath. These JAR files are needed to compile Web module:

a. From the Navigator View in WebSphere Studio Application Developer, select the **OrderEntryWeb** folder. Right-click it and select **Properties**.

b. In the Properties window, select **Java Build Path**.

c. Click the **Libraries** tab.

d. Click the **Add External Jars** button.

e. Navigate to **%WAS_HOME%\lib** and select **brbClient.jar**, **aswa.jar**, **i18nctx.jar**, and **ras.jar**.

f. Click **OK** in the Properties window to save the changes.

4. Add **aswa.jar** to Deployed_OrderEntryEJB project:

a. Right-click **Deployed_OrderEntryEJB**.

b. In the **Properties** window, select **Java Build Path**.

c. Click the **Libraries** tab.

d. Click the **Add External Jars** button.

e. Navigate to **%WAS_HOME%\lib** and select **aswa.jar**.

f. Click **OK** in the Properties window to save the changes.

g. At this point, you should see no error (ignore warnings).

5. Install the OrderEntry application in WebSphere Application Server Enterprise Edition:

a. Export the OrderEntry project from Application Developer:

i. From the Navigator View in WebSphere Studio Application Developer, select **OrderEntry** folder. Right-click it and select **Export EAR file**.

ii. Click the **Browse** button and navigate to **%OE_HOME%**.

iii. Type the name of the EAR file, which is `OrderEntryWithBRBeans.ear`.

iv. Select the **Export source files** box.

v. Click **Finish**.

b. From the WebSphere Administrative Console, start the **Install Enterprise Application** wizard.

c. Select the **Install Application** radio button.

d. Select **%OE_HOME%\OrderEntryWithBRBeans.ear**.

e. Proceed through the wizard and click **Finish**.

f. Select **No** in the Regenerate Code dialog box.

6. Import the BRBeans rule configurations file. The file contains the configuration for the BRBeans rules provided with the OrderEntry sample application:

   a. In the WebSphere Administrative Console, start **OrderEntryServer**.

   > **Important:** The Rule Management Application needs to access the Naming service. As a result, you have to start an application server prior to starting the Rule Management Application.

   b. Start the Rule Management Application:

      i. Open a command prompt and change the current directory to `%WAS_HOME%\Enterprise\bin`.

      ii. Issue the following command:

      ```
      rulemgmt %OE_HOME%\BRBeans\brbeansOrderEntryProperties
      ```

   > **Important:** The brbeansOrderEntryProperties file contains WebSphere server host and port information, as well as the JNDI names of BRBeans EJBs. If you are accessing the WebSphere server remotely, modify the host and port information before using the Rule Management Application. BRBeans EJBs JNDI names specified in the properties file must match the JNDI names used in the BRBeans configuration in step vii. on page 20.

   c. In the Rule Management Application, select **File-> Import**.

   d. Select the rule configuration file; browse to **%OE_HOME%\BRBeans** and select **OrderEntryBRBeansRules.xml**.

   e. Click **OK**.

   After you import the XML file, the Rule Management Application should look like the example shown in Figure 3-3.

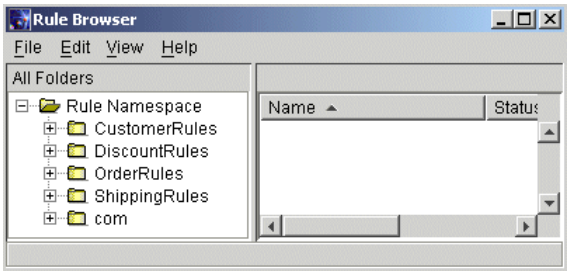


Figure 3-3 Importing rule configurations in the Rule Management Application

3.2.2 General approach for implementing business rules

The process of implementing a business rule (classifier or base) in the BRBeans framework consists of three steps:

1. Creating a rule implementor class. This step is optional; developers can use rule implementors provided with the BRBeans framework.
2. Configuring a BRBeans rule in the Rule Management Application.
3. Placing a call to the BRBeans rule in the rule client (trigger point).

These activities can be carried out by different individuals at different times. Developers are responsible for creating rule implementors and placing trigger points in the rule client. BRBeans rules can be configured in the Rule Management Application by either a domain expert or a developer.

Creating rule implementors

Rule implementors are Java classes that encapsulate business logic. These classes must implement the RuleImplementor interface. The interface has three methods:

- *init*: This method provides the initial state to the rule implementor. The method is called by the framework when a BRBeans rule is first fired. Rule implementor initialization parameters are configured in the Rule Management Application.
- *fire*: This method implements business logic and returns results of an algorithm to the caller. The `TriggerPoint` object invokes the *fire* method from the rule client passing parameters expected by the rule implementation.
- *getDescription*: This method returns a string containing a description of the RuleImplementor.

Configuring business rules

The Rule Management Application is used to configure and maintain BRBeans rules. The tool gives the ability to perform the following tasks:

- Create a new BRBeans rule definition
  - BRBeans rule folder and name (used to look up the BRBeans rule by the `TriggerPoint` object)
  - BRBeans rule start and end date
  - BRBeans rule type: Classifier, non-classifier, or classified
  - BRBeans rule implementor Java class
  - Initialization parameters for the rule implementor
  - Dependent BRBeans rules
- Modify a BRBeans rule definition
- Export or import BRBeans rule definitions, which are created in XML format.

Using Quick Copy in the Rule Management Application

The Rule Management Application provides the Quick Copy function to replace the existing BRBeans rule with a new one on a specified date. The Quick Copy function creates a copy of the existing rule and allows you to change the BRBeans rule start date and the initialization parameters. Once the copy of the BRBeans rule is created, the end date of the original rule is set to the start date of the new rule.

Note: The BRBeans suggested method for modifying existing rules is to expire the old BRBeans rule and establish a new BRBeans rule to take effect simultaneously. Using this methodology, old rules are never deleted. They are retained for historical purposes, so that if a BRBeans rule is fired with a reference date at some time in the past, the correct logic is applied.

To use Quick Copy, follow these steps:

1. Select a BRBeans rule in the Rule Management Application.
2. Right-click the rule and select **Quick Copy** (see Figure 3-4).

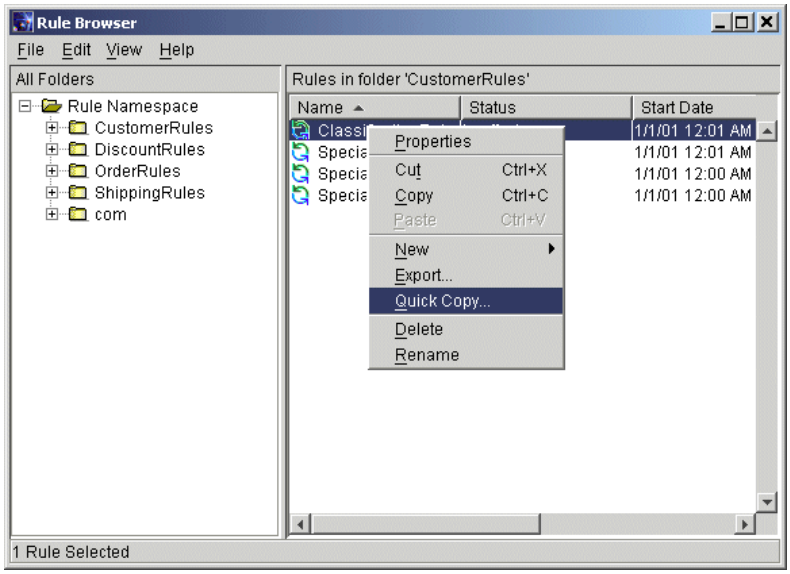


Figure 3-4 Using Quick Copy

Placing trigger points

Trigger points are pieces of application code that make a call to a BRBeans rule. Trigger points can be placed in EJBs, servlets, or Java classes that depend on business logic implemented in BRBeans rules. The `TriggerPoint` object looks up BRBeans rules by names defined in the Rule Management Application.

Using strategy objects

At runtime, the `TriggerPoint` object performs a sequence of tasks to fire a BRBeans rule. The tasks include finding, filtering, firing a BRBeans rule, and returning rule results. Strategies are Java objects provided with the BRBeans framework that influence the execution of *TriggerPoint* methods. Four strategies can be used to modify behavior of the `TriggerPoint` object (see Figure 3-5).

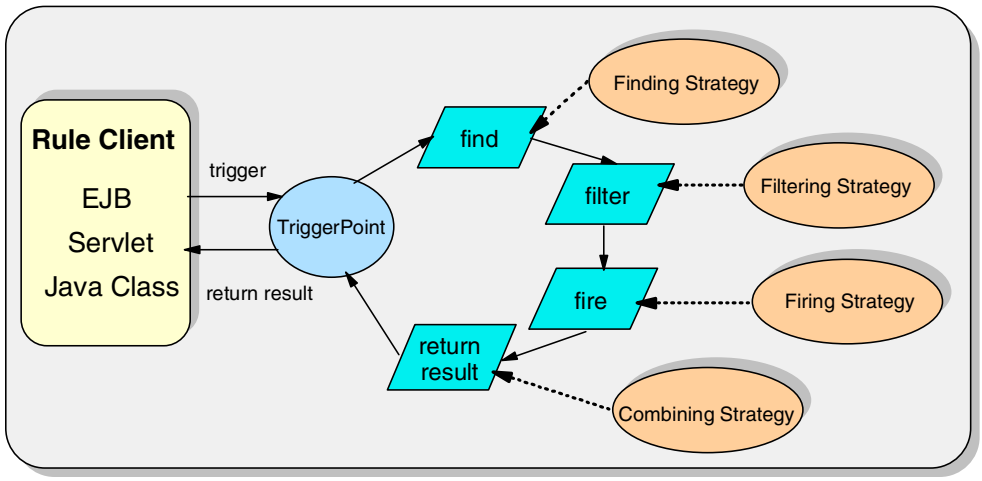


Figure 3-5 Using strategy objects in the BRBeans framework

The strategy objects are:

- **Finding strategy**: Two implementations of this strategy, DefaultClassifierFindingStrategy and DefaultNonClassifierFindingStrategy, perform a search for classifier and base rules,

respectively. The order in which BRBeans rules are found is determined by the rule precedence. The rule with the highest precedence (lowest value) is found first, and the one with the lowest precedence is found last. The rule precedence is specified in the Rule Management Application on the Other tab of the Properties window.

- **Filtering strategy**: The finding strategy can return more than one BRBeans rule. Several implementations of the filtering strategy provide different ways to filter rules found by the framework:
  - *Accept Any*: Any number of rules returned by the finding strategy will be used (default)
  - *Accept One*: One rule is expected to be found
  - *Accept First*: Only the first rule found by the framework will be fired
  - *Accept Last*: Only the last rule found by the framework will be fired
- **Firing strategy**: The firing strategy is responsible for invoking the BRBeans rules returned by the filtering strategy. The framework provides one firing strategy because all rules are fired the same way.
- **Combining strategy**: Implementations of the combining strategy provide different ways to handle BRBeans rule results:
  - *Return All*: The results of all fired rules are returned (default).
  - *Return First*: The result of the first fired rule is returned.
  - *Return Last*: The result of the last fired rule is returned.
  - *Return AND*: Returns the logical AND of the fired rule results. All fired rules must return `ConstraintReturn` or Boolean objects.
  - *Return OR*: Returns the logical OR of the results from the fired rules. All fired rules must return `ConstraintReturn` or Boolean objects.
  - *Throw Violation*: Returns a true `ConstraintReturn` object if all fired rules are successful, and throws `ConstraintViolationException` if some of the fired rules fail.

As in the finding strategy, the result of the BRBeans rule with the highest precedence is considered to be the first result.

Note: Developers can create custom strategies if the predefined BRBeans strategies do not provide the functionality required by the application. For more information on creating custom strategies, see the BRBeans product documentation at:
`http://www.ibm.com/software/webservers/appserv/doc/v40/aee/index.html`

BRBeans rule states

BRBeans rule state determines weather a rule can be used by the rule client at runtime. A BRBeans rule can be in one of five states:

- **In effect**: The BRBeans rule is active (start date is before the current date and time, and the end date is either blank or after the current date and time) and is available for use. BRBeans rules must be in this state in order to be fired from the rule client.
- **Scheduled**: BRBeans rule start date is set to some date in the future.
- **Expired**: The BRBeans rule is no longer active (rule end date is in the past).
- **Invalid**: The BRBeans rule configuration has errors.
- **Unavailable**: The BRBeans rule is not ready for use.

Rule states can be modified in the Rule Management Application.

Note: For more details on changing a rule state, see the BRBeans product documentation at: http://www.ibm.com/software/webservers/appserv/doc/v40/aee/index.html

BRBeans rule firing locations

The firing location determines where the rule implementor is instantiated and invoked. The firing location is specified in the Rule Management Application (Implementation tab of the Properties window). There are three possible values for the firing location:

- **Local**: The rule implementor is fired in the same JVM as the rule client.
- **Remote**: The rule implementor is invoked on the server where the rule implementor is installed.
- **Anywhere**: The BRBeans framework first tries to fire the rule implementor locally. If the rule implementor is not found, the framework tries to fire it remotely.

Important: We use the local firing location in the examples provided in this chapter. If you install the OrderEntry sample application on the remote WebSphere server, change the rule firing location to *remote*.

3.2.3 Implementing a classifier rule

We start the process of developing the business rules by defining a classifier rule that we will use in the OrderEntry application.

Classifier rule: If a customer has purchased X dollars worth of products YTD, then the customer qualifies for Y level.

We follow the general guideline defined in 3.2.2, “General approach for implementing business rules” on page 22.

Creating a rule implementor class

1. Create a rule implementor (see the complete class implementation in Example 3-15 on page 48).
   a. In the **Navigator** view of WebSphere Studio Application Developer, expand the **OrderEntryWeb** folder.
   b. Select the **source\rules** folder and right-click.
   c. Select **New-> Other**.
   d. Select **Java** in the left pane of the pop-up window and **Java class** in the right pane.
   e. Type the name of the class, which is `CustomerClassification`.
   f. Click the **Add** button next to the Implements Interfaces box.
   g. Select the **RuleImplementor** interface.
   h. Click **Finish**.
2. Implement the *RuleImplementor* methods:

   In the **Navigator** view of WebSphere Studio Application Developer, expand the **OrderEntryWeb\source\rules** folder. Double-click **CustomerClassification**, and implement the methods that are explained here:

   – *init*: The *init* method expects three parameters:
     - The amount of YTD purchases for the Preferred level
     - The amount of YTD purchases for the Gold level
     - The amount of YTD purchases for the Platinum level

These initialization parameters are specified in the Rule Management Application. They are passed as an array of `java.lang.Objects` (Object[] parms) and used by the *fire* method to determine customer classification. Example 3-2 shows the *init* method implementation.

Example 3-2 Implementation of the init method

```
public void init(Object[] parms,
                 String[] dependentRules,
                 String userDefinedData,
                 IRuleCopy rule) throws BusinessRuleBeansException {

    // Three parameters are expected
    ImplementorHelper.assertParamLength(parms, 3, "CustomerClassification.init");

    // Store the initialization parameters.
    preferredLevel = ((Integer) parms[0]).intValue();
    goldLevel = ((Integer) parms[1]).intValue();
    platinumLevel = ((Integer) parms[2]).intValue();

    initialized = true;
}
```

– *fire*: This method expects one parameter – the amount of YTD purchases. The amount of YTD purchases is compared to the level amounts, and the resulting classification is returned as `java.lang.String`. Implementation of the *fire* method is shown in Example 3-3.

Example 3-3 Implementation of the fire method

```
public Object fire(TriggerPoint tp,
                   Object target,
                   IRuleCopy rule,
                   java.lang.Object[] parms) throws BusinessRuleBeansException
   {
      String classification = null;
      int amount = 0;

      if (initialized){
         // One parameter is expected - the amount of customer purchases YTD
         ImplementorHelper.assertParamLength(parms, 1, "CustomerClassification.fire");
         amount = ((Integer) parms[0]).intValue();
         if(amount >= platinumLevel){
            classification = PLATINUM_LEVEL;
         }else if(amount >= goldLevel){
            classification = GOLD_LEVEL;
         }else if(amount >= preferredLevel){
            classification = PREFERRED_LEVEL;
         }else{
            classification = LEVEL_NA;
         }
      }
      return classification;
}
```

– *getDescription*: This method returns the method description.

3. Save the `CustomerClassification` class.

The complete code for the rule implementation is packaged inside the %OE_HOME%\OrderEntry.ear file. You can also see the source code for CustomerClassification.java in Example 3-15 on page 48.

Note: If you did not implement the CustomerClassification rule class, import the class into the OrderEntry project. The rule will be used in other examples described later in this chapter.

Configuring a BRBeans rule

4. Configure the CustomerClassification rule in the Rule Management Application:
   a. Make sure that WebSphere Default Application Server is running.
   b. Start the Rule Management Application if it is not already started (see 3.2.1, “Setting up the development environment” on page 19).
   c. In the Rule Management Application, expand the **Rule Namespace** folder and select the **CustomerRules** folder. The Rule Management Application should look like the example shown in Figure 3-6.

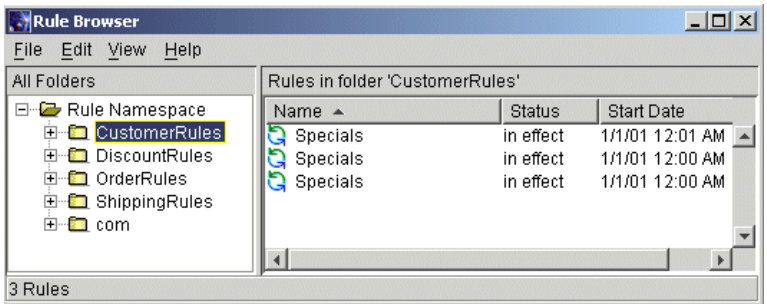


Figure 3-6 Configuring the CustomerClassification rule in the Rule Management Application

   d. Right-click the **CustomerRules** folder.
   e. Select **New-> Rule**.
   f. On the **General** tab, complete the required fields and actions (see Figure 3-7):
      i. For Name, enter: `CustomerClassification`.
      ii. For Start Date, enter: `01/01/01 12:01 AM`.
      iii. Select the **Rule performs a classification** radio button.

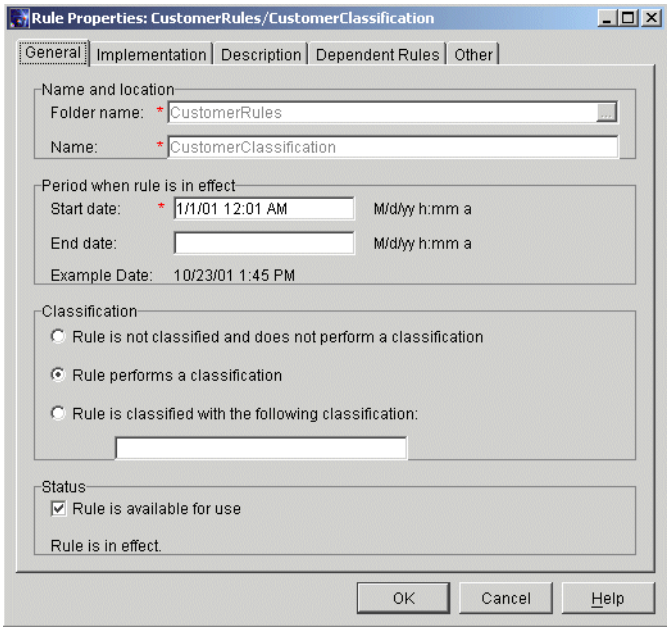


Figure 3-7 General properties of the CustomerClassification rule

g. Click the **Implementation** tab.
h. Click the **Add** button next to the initialization parameters list and add three parameters:
   - Amount to qualify for the Preferred Club:

     Type: Integer
     Value: 1000
   - Amount to qualify for the Gold Club:

     Type: Integer
     Value: 2000
   - Amount to qualify for the Platinum Club

     Type: Integer
     Value: 3000
i. Select the Local firing location (see Figure 3-8).

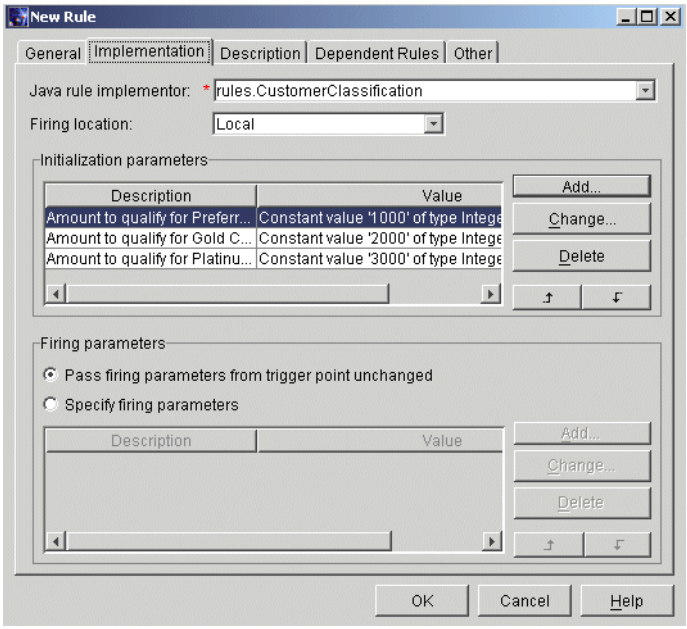


Figure 3-8 Implementation properties of the CustomerClassification rule

j. Click **OK**.

k. The CustomerClassification rule should now be in effect (see Figure 3-9).

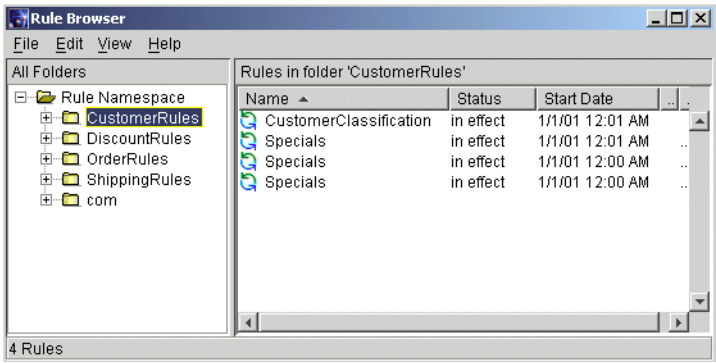


Figure 3-9 Displaying CustomerClassification status

Implementing a trigger point

5. Implement a trigger point in the rule client:

   a. In the **Navigator** view of WebSphere Studio Application Developer, expand the **OrderEntryWeb\source\rules** folder.

   b. Double-click **ApplicationMgr**.

   c. Implement the *getCustomerLevel* method. The method creates an instance of the `TriggerPoint` object, populates the required parameters, fires the rule, and processes the result returned by the CustomerClassification rule. Source code for the *getCustomerLevel* method is shown in Example 3-4.

> **Note:** Caching is disabled in the rule client implementations provided in this chapter for demonstration purposes only. For more information about caching, see “Caching” on page 46.

Example 3-4 Implementation of the getCustomerLevel method

```
public void getCustomerLevel(int amountYTD) throws Exception{

   // Create new trigger point
   TriggerPoint tp = new TriggerPoint();

   // Disable caching - for demonstration purpose
   tp.disableCaching();

   // The rule expects one parameter: the amount of YTD purchases to the rule
   Object[] firingParams = { new Integer(amountYTD) };
   String ruleName = "CustomerRules/CustomerClassification";

   // Call the rule
   Object result =
      tp.triggerClassifier(null, // Target Object - not reqiured by rule implementation
                           firingParams, // Rule Implementor firing parameters
                           ruleName);    // The name of the rule to fire

   if (result == null ) {
      itemsViewBean.setApplicationError("The rule returned null");
   }
   else {
      // Rule successfully called
      Object[] resultArray = (Object[]) result;
      // The rule returns one String: customer level
      customerLevel = (String) resultArray[0];
   }

   // Save customer level
   itemsViewBean.setCustomerLevel(customerLevel);
}
```

d. Save the changes to the `ApplicationMgr` class.

6. Save all opened files and export the OrderEntry EAR project from WebSphere Studio Application Developer to %OE_HOME%.
7. Name the file `OrderEntryWithRules.ear`.
8. Install **OrderEntryWithRules.ear** in WebSphere Application Server Enterprise Edition:

> **Attention:** Prior to installing OrderEntryWithRules.ear, uninstall the previously installed OrderEntry project in WebSphere.

   a. Start WebSphere Administrative Console.
   b. From the WebSphere Administrative Console, start the **Install Enterprise Application** wizard.
   c. Select the **Install Application** radio button.
   d. Browse to **%OE_HOME%**.
   e. Select **OrderEntryWithRules.ear**.
   f. Proceed through the wizard.
   g. Click **Finish**.
   h. On the Generate Code dialog, click **No**.

9. Test the application:

a. In the WebSphere Administrative Console, start **OrderEntryServer**.
b. Bring up the application home page:

```
http://localhost:<port_number>/OrderEntry/index.html
```

c. Select **New Order**.
d. Type the customer ID `0007` and click the **Submit** button.
e. The next page displays customer level information:

Customer ID: 0007	**Name: Jim J Fair**	**Customer Level: Platinum**

With the Rule Management Application, you can change the amounts required to qualify for a certain customer level (rule initialization parameters) and see the changes reflected in the application without recompiling application code.

a. Start the Rule Management Application, if it is not already started (see 3.2.1, “Setting up the development environment” on page 19).
b. Expand the **Rule Namespace** folder, and select the **CustomerRules** folder.
c. Right-click the **CustomerClassification** rule, and select **Properties**.
d. Select the **Implementation** tab.
e. Select the initialization parameter for the Platinum level, and click **Change**.
f. Change the value to `20000`.

> **Note:** In the production environment, we recommend that you use Quick Copy (see “Using Quick Copy in the Rule Management Application” on page 23) to change the rule initialization parameters.

g. Test the rule (see step 7). Use the same ID as in the first test – `0007`. The customer level has changed.

Customer ID: 0007	**Name: Jim J Fair**	**Customer Level: Gold**

You have successfully implemented a simple classifier rule. We will use the same three-step process to implement more complex rules described later in this chapter:

- Creating a rule implementor
- Configuring a BRBeans rule in the Rule Management Application
- Placing a trigger point in the rule client

3.2.4 Implementing base rules

As we examine the implementations of base business rules, we review the use of strategies in the BRBeans framework. This section discusses the implementation of this business rule.

> **Base rule**: Customers can get discounts based on a customer level, the total amount of purchase, and other factors (holiday discounts, promotions on certain items, and so on). Discount values and specials may change frequently.

The rule definition states that a customer discount is a cumulative value derived from several factors. We need to implement the rule in a way that allows us to add new discount factors and change discount values. To achieve this flexibility, we implement each discount factor as a separate rule and provide discount values as rule initialization parameters. We can add new discount factors by creating new rule implementors and configuring them in the Rule Management Application. We can also change discount values in the Rule Management Application. Neither of these tasks requires changes in the rule client.

Base rules implementation

In WebSphere Studio Application Developer, expand the **OrderEntryWeb** folder. In the source\rules folder, you can find implementations of two base rules:

- DiscountByLevel
- DiscountByTotal

DiscountByLevel implementation returns a discount percentage based on a customer level. Rule implementor initialization parameters include amounts required to qualify for each customer level and discount values for each customer level. Implementation of the *fire* method is shown in Example 3-5 (implementation of the *init* method is similar to Example 3-2 on page 27).

Example 3-5 Implementation of the DiscountByLevel fire method

```
public Object fire(TriggerPoint tp,
              Object target,
              IRuleCopy rule,
              java.lang.Object[] parms) throws BusinessRuleBeansException
{
    int discount = 0;
    int amountYTD = 0;

    if (initialized){
        // In all discount rules, the first parameter is the amount of purchase
        // The amount of purchase is not used by this rule
        // The second parameter in the array is the amount of YTD purchases
        amountYTD = ((Integer) parms[1]).intValue();
        // Level and discount variables were initialized in init()
        if(amountYTD >= platinumLevel){
            discount = platinumDiscount;
        }else if(amountYTD >= goldLevel){
            discount = goldDiscount;
        }else if(amountYTD >= preferredLevel){
            discount = preferredDiscount;
        }
    }
    return new Integer(discount);
}
```

DiscountByTotal returns a discount percentage based on the amount of purchase. Rule implementor initialization parameters include three levels of purchase amount required to qualify for a certain discount and discount values for each level. Business logic is implemented in the *fire* method as shown in Example 3-6.

Example 3-6 Implementation of the DiscountByTotal fire method

```
   public Object fire(
      TriggerPoint tp,
      Object target,
      IRuleCopy rule,
      java.lang.Object[] parms)
      throws BusinessRuleBeansException {
      int discount = 0;
      int amount = 0;

      if (initialized) {
```

```
            // The first parameter is the amount of purchase
            amount = ((Integer) parms[0]).intValue();

            // The customer's amount of purchase is compared against 3 different values
            // to figure out a discount.
            // The 3 levels of the discounts with corresponding "eligible purchase"
            // amounts are defined through the Rule Management application.

            if (amount >= total3) {
               discount = level3Discount;
            }
            else if (amount >= total2) {
               discount = level2Discount;
            }
            else if (amount >= total1) {
               discount = level1Discount;
            }
         }
         // The method can return 0
         return new Integer(discount);
}
```

Base rules configuration

Start the Rule Management Application if it is not already started (see 3.2.1, “Setting up the development environment” on page 19).

Expand the **Rule Namespace** folder and select the **DiscountRules** folder. You should see two CalculateDiscount rules as shown in Figure 3-10.

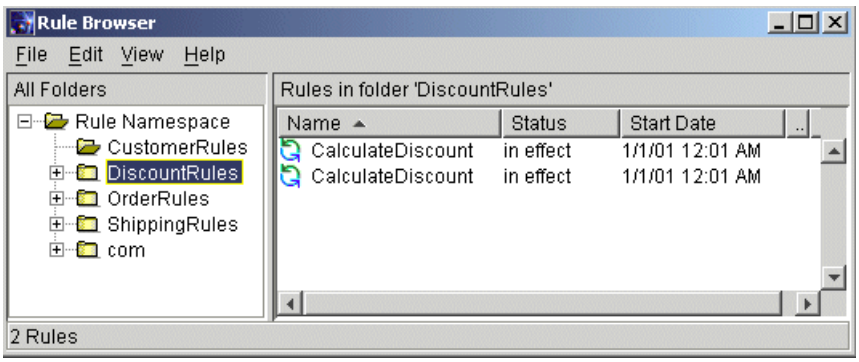


Figure 3-10 Discount rules configuration

Examine the properties of each rule:

1. Select the rule and right-click. Select **Properties**.
2. The **General** tab is identical for both rules. Notice that the first radio button, **Rule is not classified and does not perform classification**, is selected. All base rules must select this classification.
3. Click the **Implementation** tab. The rules have different implementation classes and different initialization parameters. One rule specifies DiscountByLevel as the implementation class, and the other one is DiscountByTotal.

Rule client implementation

The next step is to implement the client's call to the rules. In WebSphere Studio Application Developer, open the **ApplicationMgr** class (in the **OrderEntryWeb\source\application** folder) and navigate to the *getOrderDiscount* method. The method performs the following tasks:

- Creates an instance of the trigger point object.
- Populates the required parameters.
- Fires the BRBeans rule CalcuateDiscount located in the DiscountRules folder.
- Processes results returned by the rule.

As you can see in Example 3-7, the rule client expects results from multiple BRBeans rules to be returned. Discount percentages returned from different BRBeans rules are added together to form a cumulative percentage discount. In this example, multiple rules can be returned because we used the default trigger point strategies (see "Using strategy objects" on page 24).

Example 3-7 Implementation of the getOrderDiscount method

```
public void getOrderDiscount(int purchaseTotal,int amountYTD)throws Exception{

   int orderDiscount  = 0;

   //Create new trigger point
   TriggerPoint tp = new TriggerPoint();

   // Disable caching - for demonstration purpose
   tp.disableCaching();

   //Pass the purchase total and amount of YTD purchases to the rule
   Object[] firingParams = { new Integer(purchaseTotal), new Integer(amountYTD) };

   Object result = tp.trigger(null, firingParams, "DiscountRules/CalculateDiscount");
   if (result == null ) {
      itemsViewBean.setApplicationError("The rule returned null");
   }
   else {
      //Rule successfully called
      Object[] resultArray = (Object[]) result;
      // Multiple discount rules might have been found
      for(int i=0;i<resultArray.length;i++){
         Integer discountValue = (Integer)resultArray[i];
         orderDiscount = orderDiscount + discountValue.intValue();
      }
   }

   itemsViewBean.setOrderDiscount(orderDiscount);
}
```

The default finding strategy, DefaultNonClassifierFindingStrategy, returns all non-classifier rules that match the "DiscountRules/CalculateDiscount" name (we configured two rules with this name). The default filtering strategy is Accept Any; all of the rules that are found stay eligible for firing. The default combining strategy is Return All; the results of all rules fired by the BRBeans framework are returned (two rules in this example).

Testing the application

To see the results of the rule implementation, test the application:

1. Export and install the OrderEntry application if necessary.
2. In the WebSphere Administrative Console, start **OrderEntryServer**, if it is not already started.
3. Bring up the application home page:

   `http://localhost:<port_number>/OrderEntry/index.html`
4. Select **New Order**.
5. Type the customer ID, `0002`, and click the **Submit** button.

   You can see the discount value based on customer level (see Figure 3-11).
6. Add several items:
   - Item ID: 000017, quantity: 1
   - Item ID: 000013, quantity: 1
   - Item ID: 000002, quantity: 1
7. As you add the items, you can see the changes in the total order discount value. The total order discount is the sum of the customer level discount and the amount of purchase discount.

   Purchases that are greater than $50 get a 5% discount; if they are greater than $75, they get a 7% discount; and if they are greater than $100, they get a 10% discount. In this example after adding the first item, the total order discount is 7% (Gold level discount); after adding the second item, the total discount is 12% (the sum of 7% Gold level discount and 5% purchase discount); and after adding the third item, the total discount is 17% (the sum of 7% Gold level discount and 10% purchase discount). See Figure 3-11.

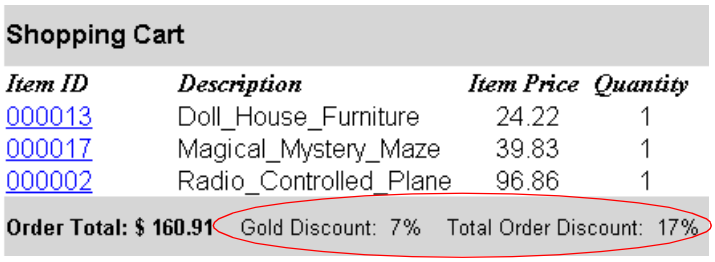


Figure 3-11 Testing discount rules implementation

This implementation of the discount rule gives us the flexibility to change the discount percentage without changing the rule client. Perform these tasks and test the application after each change:

8. Expire the second discount rule in the Rule Management Application by changing the end date in the Properties window to some date in the past. The status of the modified BRBeans rule is expired, as shown in Figure 3-12.

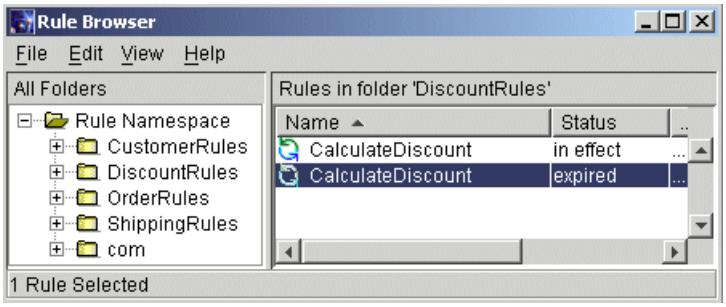


Figure 3-12 Status of the CalculateDiscount rules after the end date change

Total order discount percentage changes because only one of the BRBeans rules, the rule that uses the DiscountByLevel rule implementor, is found and fired (see Figure 3-13). The discount percentage does not change as we add more items because the customer receives the level discount only (7% Gold discount).

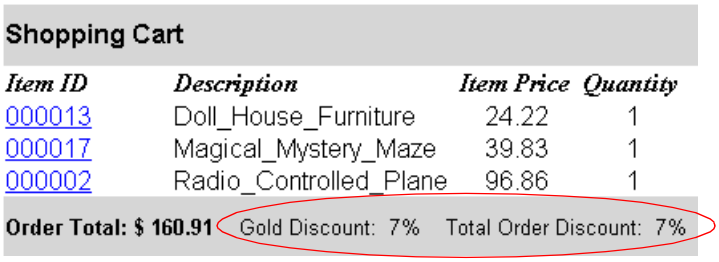


Figure 3-13 Total order discount after inactivating one of the discount rules

9. Change the discount value or the level required to qualify for a certain discount in one or both discount rules. The discount percentages are specified as initialization parameters in the Rule Management Application. They can be changed in the **Implementation** tab of the rule Properties window.

10. Add another rule implementation, such as HolidayDiscount, that simply returns a discount passed in initialization parameter. The *fire* method of this BRBeans rule should return `java.lang.Integer`.

    Configure the HolidayDiscount rule in the Rule Management Application with the same name as other discount rules, CalculateDiscount. When you run the application, the discount percentage changes because the discount value is a sum of the discounts returned from three BRBeans rules.

Using trigger point strategies

To get a better understanding of the use of strategies in the BRBeans framework, modify the filtering or the combining strategy of the `TriggerPoint` object. For example, the *getFirstCustomerDiscount* method in `ApplicationMgr` uses Return First combining strategy (see Example 3-8).

When the `TriggerPoint` object is configured with this strategy, the results of the first rule matching the specified name (DiscountRules/CaclulateDiscount) are returned. Since the rule is guaranteed to return one result only, we handle the result object differently than in the *getOrderDiscount* method. In the latter case, the results of the rule with DiscountByLevel implementation are returned because this rule has the highest precedence. The BRBeans rule precedence is specified on the Other tab of rule Properties window in the Rule Management Application.

Example 3-8 implementation of getFirstCustomerDiscount method

```
public void getFirstCustomerDiscount(int purchaseAmount, int amountYTD) throws Exception{

   // Create new trigger point
   TriggerPoint tp = new TriggerPoint();

   // Disable caching - for demonstration purpose
   tp.disableCaching();

   // Set the combining strategy to return the first
   // The second parameter specifies that this combining strategy will be applied to
   // all rules - classifier and base
   tp.setCombiningStrategy(CombiningStrategy.RETURN_FIRST,tp.ALL_RULES);

   // Pass the purchase total and amount of YTD purchases to the rule
   Object[] firingParams = { new Integer(purchaseAmount), new Integer(amountYTD) };

   Object result = tp.trigger(null, firingParams, "DiscountRules/CalculateDiscount");
   if (result == null ) {
      itemsViewBean.setApplicationError("The rule returned null");
   }else{
      // Rule successfully called
      customerDiscount = ((Integer) result).intValue();
   }
   itemsViewBean.setCustomerDiscount(customerDiscount);
}
```

Testing the application

To see the results of the *getCustomerFirst* method implementation, test the application:

1. In the WebSphere Administrative Console, start **OrderEntryServer**, if it is not already started.
2. Bring up application home page:

   `http://localhost:<port_number>/OrderEntry/index.html`
3. Select **New Order** and type in customer ID, which is `0001`.
4. Click the **Submit** button.

   You can see the discount value based on the customer level. This customer has qualified for the Platinum level and receives a 10% discount.

   Order Total: $ 0 Discount: 10% Total Order Discount: 0%

5. Select **New Order**, and type in the customer ID, which is `0005`.
6. Click the **Submit** button.
7. This customer has qualified for the Gold level and receives a 7% discount.

   Order Total: $ 0 Discount: 7% Total Order Discount: 0%

Implementing classified rules

The Trigger Point framework provides a *triggerSituational* method. This method first calls a classifier rule and then uses the derived classification to invoke a base rule. The base rule in this case is called a *classified rule*; it is classified by the string returned from the classifier rule. The definition of the business rule is stated here.

Business rule definition: Customers of different levels qualify for different specials.

To implement classified rules, in WebSphere Studio Application Developer, expand the **OrderEntryWeb** folder. In the **source\rules** folder, you can find the implementation of three classified rules: **PreferredSpecials**, **GoldSpecials**, and **PlatinumSpecials**. These rule implementors return an array of specials (`java.lang.String`) that were passed to the rules as initialization parameters (see Example 3-9).

Example 3-9 Implementation of the Platinum Specials rule

```
public void init(Object[] parms,
                 String[] dependentRules,
                 String userDefinedData,
                 IRuleCopy rule) throws BusinessRuleBeansException
{
   // Save platinum level specials
   specials = parms;
}

public Object fire(TriggerPoint tp,
                   Object target,
                   IRuleCopy rule,
                   java.lang.Object[] parms) throws BusinessRuleBeansException
{
   return specials;
}
```

Rule configuration

Start the Rule Management Application if it is not already started (see 3.2.1, “Setting up the development environment” on page 19). Expand the **Rule Namespace** folder, and click the CustomerRules folder. You can see four rules for CustomerClassification and three **Specials** rules. CustomerClassification returns one of three values: Preferred, Gold, or Platinum. There is a classified Specials rule for each possible return value (see Figure 3-14).

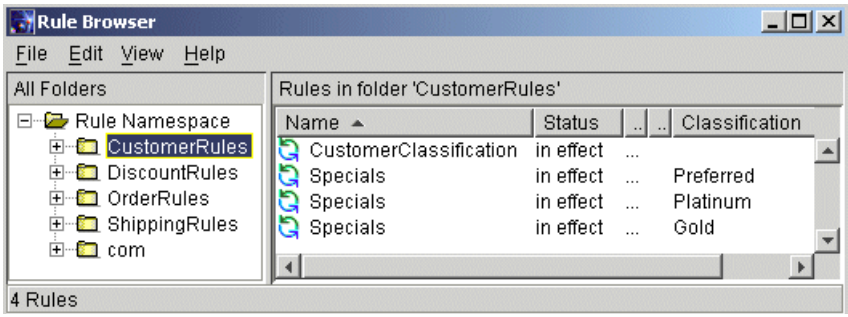


Figure 3-14 Configuring the classified rules

Open the **Properties** window for each Specials rule. In the Classification section on the General tab, each Specials rule is classified with a different string (Preferred, Gold, or Platinum). On the **Implementation** tab, notice that each rule has different implementation class and initialization parameters (specials for each level).

Rule client implementation

In WebSphere Studio Application Developer, navigate to the *getCustomerSpecials* method in the ApplicationMgr class (the OrderEntryWeb\source\application folder). The classifier rule in this method is the CustomerClassification rule we created earlier in this chapter. “Specials” is the name of the base rule (or rules) that is invoked based on the value returned from CustomerClassification.

In the *getCustomerSpecials* method (see Example 3-10), we populate the required parameters for both the classifier rule and the classified rule, fire the classifier and classified rules, and process the results. Note that only the results of the classified rule are returned.

Example 3-10 Implementation of the getCustomerSpecials method

```
// Classifier rule requires one parameter: the amount of YTD purchases
Object[] classifierParams = { new Integer(amountYTD) };
// Specials rule does not require any parameters
Object[] classifiedParams = null;
// Rule Names
String classifierName = "CustomerRules/CustomerClassification";
String classifiedName = "CustomerRules/Specials";

Object result = tp.triggerSituational(this,
                                      classifiedParams,
                                      classifierParams,
                                      classifiedName,
                                      classifierName);
if (result == null ) {
    itemsViewBean.setApplicationError("The rule returned null");
}else {
    //Rule successfully called
    Object[] resultArray = (Object[]) result;
    // The first element is the array of specials
    Object [] specials = (Object []) resultArray[0];
    for(int i =0;i<specials.length;i++){
        String special = (String)specials[i];
        custSpecials.addElement(special);
    }
}
itemsViewBean.setCustomerSpecials(custSpecials);
```

Testing the application

To see the results of rule implementations, test the application:

1. In the WebSphere Administrative Console, start **OrderEntryServer**, if it is not already started.
2. Bring up the application home page:

   `http://localhost:<port_number>/OrderEntry/index.html`
3. Select **New Order**.
4. Type in the customer ID, which is `0002`, and click the **Submit** button.

   The next page displays specials for the Gold level (see Figure 3-15).

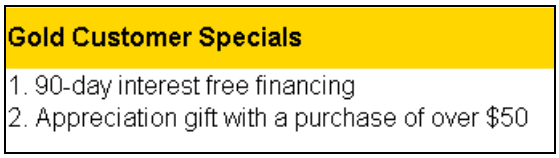


Figure 3-15 Gold customer specials

5. Repeat the described steps for customer ID 0012.

   This customer has qualified for the Preferred level. Preferred level specials are displayed as shown in Figure 3-16.

Figure 3-16 Preferred customer specials

6. In the Rule Management Application, change the specials offered to the customer (initialization parameters for the Specials rules on the Implementation tab of the Properties window). For example, you can add a new special, “Free shipping with a purchase of over $100”, for the Gold level:

   a. Start the Rule Management Application if it is not already started (see 3.2.1, “Setting up the development environment” on page 19).

   b. Expand the **Rule Namespace** folder. Click the **CustomerRules** folder.

   c. Select the **Specials** rule with the **Gold** classification. Right-click and select **Properties**.

   d. In the Properties window, select the **Implementation** tab.

   e. Click the **Add** button next to the Initialization parameters box.

   f. Fill out the required fields:

      - Description: **Gold customer special**
      - Type: **String**
      - Value: **Free shipping with a purchase of over $100**

   g. Click **OK**.

7. Test the application with the user ID 0002. The results are displayed in Figure 3-17.

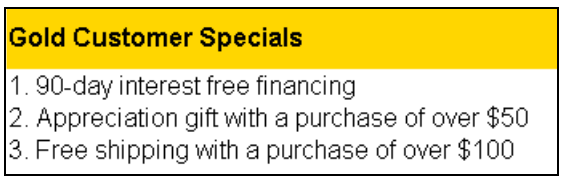

Gold Customer Specials
1. 90-day interest free financing
2. Appreciation gift with a purchase of over $50
3. Free shipping with a purchase of over $100

Figure 3-17 Modified Gold customer specials

3.2.5 Implementing a business rule with dependent rules

Some complex business processes are composed of several business rules. We can model this behavior in the BRBeans framework. We simply use one BRBeans rule to represent the process and one or more dependent BRBeans rules to represent the various units of the composition.

To demonstrate the use of dependent rules, let's review the implementation of this business rule.

The total order amount is determined by:

1. Getting the actual purchase amount
2. Subtracting discount
3. Adding the tax
4. Adding the shipping charge

Rule implementations

In WebSphere Studio Application Developer, expand the **OrderEntryWeb** folder. As in previous examples, the source code for the rules is located in the **source\rules** folder. CalculateOrderTotal is a rule implementor that encapsulates the total order amount calculation functionality. The initialization parameters for the *init* method are the names of dependent rules that are used by CalculateOrderTotal, and a sales tax rate (see Example 3-11.

Example 3-11 Implementation of the CalculateOrderTotal init method

```
public void init(Object[] parms,
                 String[] dependentRules,
                 String userDefinedData,
                 IRuleCopy rule) throws BusinessRuleBeansException
{
    // Store the initialization parameters.
    // Tax rate
    taxRate = ((Float) parms[0]).floatValue();
    // Dependent rules
    this.dependentRules = dependentRules;

    initialized = true;
}
```

CalculateOrderTotal uses two dependent rules: CalculateDiscount and CalculateShippingCharge. We reviewed the source code for the discount rules in "Base rules implementation" on page 33. CalculateShippingCharge (see Example 3-12) returns a shipping rate based on the type of shipping selected by the customer: ground, priority, or express. Shipping types and rates are specified as initialization parameters to the CalculateShippingCharge rule.

Example 3-12 Implementation of the CalculateShippingCharge fire method

```
public Object fire(TriggerPoint tp,
                   Object target,
                   IRuleCopy rule,
                   java.lang.Object[] parms) throws BusinessRuleBeansException
   {
      float shippingCharge = 0;
      String shippingType = null;

      if (initialized){
         // One parameter is expected - the shipping type
         ImplementorHelper.assertParamLength(parms, 1, "CalculateShippingCharge.fire");
         shippingType = (String) parms[0];
```

```
          if(shippingType.equals(EXPRESS)){
             shippingCharge = expressCharge;
          }else if(shippingType.equals(PRIORITY)){
             shippingCharge = priorityCharge;
          }else if(shippingType.equals(GROUND)){
             shippingCharge = groundCharge;
          }
       }

       return new Float(shippingCharge);
    }
```

The *fire* method (see Example 3-13) invokes the dependent rules, performs the calculation, and returns the total order amount. The names of the dependent rules called by the *fire* method are defined in the Rule Management Application.

Example 3-13 Implementation of the CalculateOrderTotal fire method

```
public Object fire(TriggerPoint tp,
                   Object target,
                   IRuleCopy rule,
                   java.lang.Object[] parms) throws BusinessRuleBeansException
{
    float orderTotal = 0;
    float orderAmount = 0;
    int discount = 0;
    Object[] resultArray = null;
    Object result = null;
    Float shippingCharge = null;
    String ruleName = null;

    if (initialized){
        // Purchase amount
        orderAmount = ((Float) parms[0]).floatValue();
        // Parameters to pass to dependent rules
        Object[] discountParams = (Object []) parms[1];
        Object[] shippingParams  = {parms[2]};

        // The first rule is the discount rule
        ruleName = dependentRules[0];

        // Get discount
        result = tp.trigger(null, discountParams, ruleName);
        resultArray = (Object[]) result;
        // Multiple discount rules might have been found
        for(int i=0;i<resultArray.length;i++){
            Integer discountValue = (Integer)resultArray[i];
            discount = discount + discountValue.intValue();
        }
       // Get the total of with the discount
        orderTotal = orderAmount - (orderAmount * discount/100);

        // Add tax
        orderTotal = orderTotal + (orderTotal * taxRate/100);

        // The second rule is the shipping rate rule
        ruleName = dependentRules[1];
        // Get the shipping rate
```

```
        result = tp.trigger(null, shippingParams, ruleName);
        resultArray = (Object[]) result;
        shippingCharge = (Float)resultArray[0];
        // Add shipping charge
        orderTotal = orderTotal + shippingCharge.floatValue();
    }
    return new Float(orderTotal);
}
```

Rule configuration

Start the Rule Management Application if it is not already started (see 3.2.1, “Setting up the development environment” on page 19). Expand the **Rule Namespace** folder, and click the **OrderRules** folder. Open the Properties window for the CalculateOrderTotal rule. On the General tab, the rule is classified as a base rule. Click to the **Dependent rules** tab. The order in which the rules are listed is important because the *fire* method expects the rules to be in a specified order (see Figure 3-18).

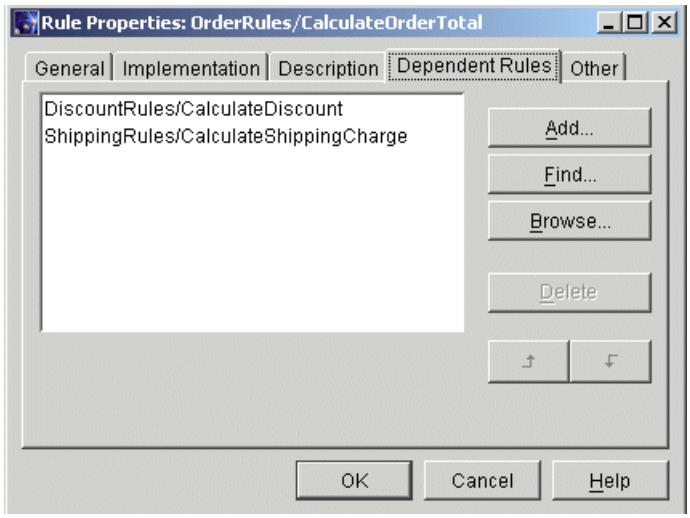


Figure 3-18 Specifying dependent rules in the Rule Management Application

Note: The BRBeans framework does not ensure that all dependent rules are fired.

Rule client implementation

In WebSphere Studio Application Developer, navigate to the *getOrderTotal* method in the `ApplicationMgr` class (the OrderEntryWeb\source\application folder). In *getOrderTotal* (see Example 3-14), we populate the required parameters for both the top-level rule and dependent rules, fire the rule, and process rule results.

Example 3-14 Implementation of the getOrderTotal method

```
public void getOrderTotal(Float purchaseTotal, Integer amountYTD, String shippingType)
throws Exception{

   Float orderTotal = null;

   //Create new trigger point
   TriggerPoint tp = new TriggerPoint();
```

```
    // Disable caching - for demonstration purpose
    tp.disableCaching();

    // This array contains parameters for the main rule and dependent rules
    Object[] firingParams = new Object[3];
    // Order amount and customer YTD purchases
    Object[] discountRuleParams = {new Integer(purchaseTotal.intValue()),amountYTD};
    // Order amount
    firingParams[0] = purchaseTotal;
    // Parameters for the discount rules
    firingParams[1] = discountRuleParams;
    // Shipping type
    firingParams[2] = shippingType;

    // Fire the rule
    Object result = tp.trigger(tp, firingParams, "OrderRules/CalculateOrderTotal");
    if (result == null ) {
       itemsViewBean.setApplicationError("The rule returned null");
    }else {
       // Rule successfully called
       Object[] resultArray = (Object[]) result;
       // The first element is the array of specials
       orderTotal = (Float) resultArray[0];
    }
    itemsViewBean.setOrderTotal(orderTotal.floatValue());
}
```

Testing the application

To see the results of the rule implementations, test the application:

1. Export and install the OrderEntry application if needed.
2. In the WebSphere Administrative Console, start **OrderEntryServer**, if it is not already started.
3. Bring up application home page:

   `http://localhost:<port_number>/OrderEntry/index.html`
4. Select **New Order**.
5. Type in the customer ID, which is `0002`, and click the **Submit** button.
6. Add several items.
7. Select the shipping type and click **Check Out**.

   The next page displays the total order amount as shown in Figure 3-19 (the discount value may be different depending on the total amount of purchase).

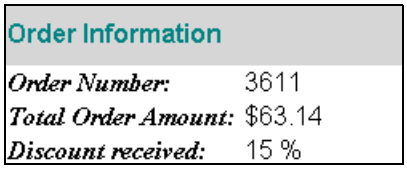


Figure 3-19 Total order amount

Other usage of dependent rules

Dependent rules can be used to encapsulate a validation process. Different validation functions, such as checking customer balance, credit limit, and item availability can be implemented as separate business rules that return an object type of Boolean or ConstraintReturn. The top-level validation rule can invoke individual validation rules and report results to the rule client.

3.3 Performance considerations

Every business rule is represented as an EJB at runtime. As a result, the BRBeans framework has to repeatedly execute two tasks to find the rule:

- Perform a query to find a rule
- Perform a remote method call to fire the rule

These operations can be costly from the performance perspective. This section discusses the techniques that can be used to improve performance.

Caching

We can eliminate the need to perform a query to find a BRBeans rule with the use of the client-side cache. The client-side cache stores the results of all queries performed to find rule EJBs. After the initial call to the BRBeans rule has been made, all subsequent calls use the rule stored in the cache. Cache scope is limited to the JVM in which the client is running.

The disadvantage of using the client-side cache is the inability to recognize BRBeans rule configuration changes immediately. A cache polling frequency determines how often the client-side cache is refreshed. The default polling frequency is 10 minutes. You can change the polling frequency in the Rule Management Application.

1. Start the Rule Management Application if it is not already started (see 3.2.1, “Setting up the development environment” on page 19).
2. Expand the **Rule Namespace** folder.
3. Expand the **com-> ibm-> websphere-> brb** folder.
4. You should see BRB CacheRule as shown in Figure 3-20.

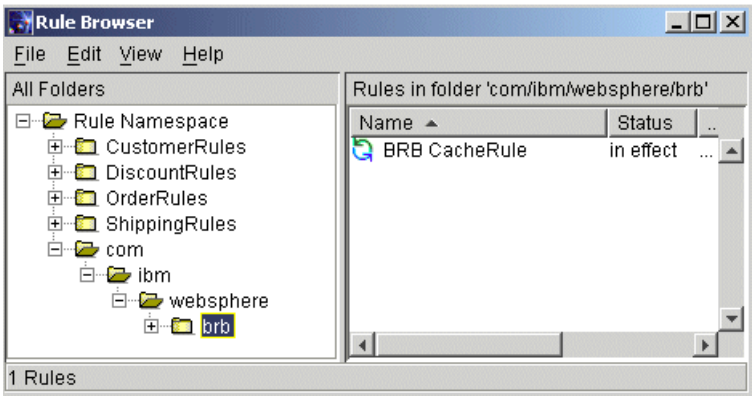


Figure 3-20 Modifying the BRB CacheRule

5. Select **BRB CacheRule**, right-click, and select **Properties**.
6. In the Properties window, select the **Implementation** tab.
7. Select the **PollFrequency** initialization parameter, and click the **Change** button.

8. Change the value of the PollFrequency initialization parameter (see Figure 3-21).
9. Click **OK**.

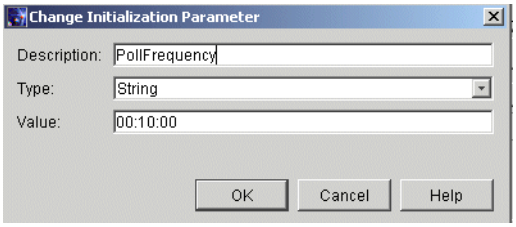


Figure 3-21 Changing the poll frequency in the Rule Management Application

Client-side caching is enabled by default.

Note: We disabled client-side caching in the examples provided with this chapter to observe the effect of changes made through the Rule Management Application without refreshing the cache.

Firing location

We can eliminate the remote call to fire a BRBeans rule by triggering the rule locally. The Firing location parameter is set on the Implementation tab of the rule Properties window in the Rule Management Application. The disadvantage of firing the rules locally is the requirement to install rule implementors on each client system that uses BRBeans rules. The best performance is achieved by using the client-side cache and local firing location.

Creating database indexes

Performance can be improved by creating an index for a rule name column in the rule definition table. In the BUSRULES database provided with OrderEntry application, business rule data is stored in the BRBEANS_RULE table. Creating an index for the rulename column in this table can improve performance of queries searching for a rule name.

Note: For more information on improving performance, see the product documentation at: http://www.ibm.com/software/webservers/appserv/doc/v40/aee/index.html

3.4 Deployment

The BRBeans framework uses a database to persist BRBeans rule data. You can create a separate database to maintain rule data or create tables in the main database used by the application. All EJBs accessed in the same transaction must specify the same isolation level. Using a different database for business rules removes this restriction because application EJBs and rule EJBs are accessed in different transactions. In our example, we used a separate database to maintain business rule data – BUSRULES.

BRBeans should be deployed as a part of the application EAR file. We completed this step while setting up BRBeans development environment (see 3.2.1, “Setting up the development environment” on page 19).

3.5 Implementing the CustomerClassification class

Example 3-15 shows the complete implementation of the `CustomerClassification` class.

Example 3-15 Implementation of the CustomerClassfication class

```
package rules;

import com.ibm.websphere.brb.RuleImplementor;
import com.ibm.websphere.brb.ConstraintReturn;
import com.ibm.websphere.brb.TriggerPoint;
import com.ibm.websphere.brb.BusinessRuleBeansException;
import com.ibm.websphere.brb.mgmt.IRuleCopy;
import com.ibm.websphere.brb.implementor.ImplementorHelper;


public class CustomerClassification implements RuleImplementor {

   // These variables are initialized in init()
   int preferredLevel;
   int goldLevel;
   int platinumLevel;
   // set to true at the end of init()
   boolean initialized  = false;

   private final static String PREFERRED_LEVEL = "Preferred";
   private final static String GOLD_LEVEL = "Gold";
   private final static String PLATINUM_LEVEL = "Platinum";

   public CustomerClassification() {
      super();
   }

   public Object fire(TriggerPoint tp,
                      Object target,
                      IRuleCopy rule,
                      java.lang.Object[] parms) throws BusinessRuleBeansException
   {
      String classification = null;
      int amount = 0;

      if (initialized){
         // One parameter is expected - the amount of customer purchases YTD
         ImplementorHelper.assertParamLength(parms, 1, "CustomerClassification.fire");
         amount = ((Integer) parms[0]).intValue();
         if(amount >= platinumLevel){
            classification = PLATINUM_LEVEL;
         }else if(amount >= goldLevel){
            classification = GOLD_LEVEL;
         }else if(amount >= preferredLevel){
            classification = PREFERRED_LEVEL;
         }
      }

      return classification;

   }

   public String getDescription() {
```

```
        return "Returns customer classification";
    }

    public void init(Object[] parms,
                     String[] dependentRules,
                     String userDefinedData,
                     IRuleCopy rule) throws BusinessRuleBeansException
     {

        // Three parameters are expected
        ImplementorHelper.assertParamLength(parms, 3, "CustomerClassification.init");

        // Store the initialization parameters.
        preferredLevel = ((Integer) parms[0]).intValue();
        goldLevel = ((Integer) parms[1]).intValue();
        platinumLevel = ((Integer) parms[2]).intValue();

        initialized = true;

    }
}
```

4

Extended Messaging Service

This chapter discusses the Extended Messaging Service (JMS Listener) and the main components of the JMS messaging implementation within WebSphere Application Server Enterprise Edition. It introduces and discusses the *point-to-point messaging model*. This model is applied to the OrderEntry application to illustrate real-world use. This chapter also includes sample code and a configuration using WebSphere Application Developer, the JMSAdmin tool, MQSeries Explorer, and the Administration Console of WebSphere Application Server, along with the expected results. Plus it introduces and explains the *publish/subscribe messaging model*.

Note: To understand the structure of the OrderEntry application and to run it, read the instructions in Appendix A, “Downloading and installing the OrderEntry application” on page 221.

© Copyright IBM Corp. 2002

4.1 An overview of basic JMS and JMS Listener

Basic JMS support allows Java applications to communicate asynchronously using JMS APIs. Communicating applications do not need to run at the same time or have any awareness of each other; they are *loosely coupled*.

JMS is a service-oriented API specification. That is, the JMS API prescribes messaging functionality in terms of interfaces, which JMS vendors then implement. Programmers work with JMS through these interfaces. Although the JMS specification is still new, it has been implemented by several vendors, including IBM, and has been endorsed by many others.

One example of JMS could be a retail company taking orders for products over the Internet and using JMS to communicate with a supplier to re-order stock as items are sold. When the stock arrives, a separate warehousing company can use JMS to update the recorded stock level of the retail company.

There is, though, a shortcoming with the basic JMS support; an application has to implement some mechanism for polling for the messages on the incoming queue. WebSphere Application Server Enterprise Edition solves this problem with the extended messaging support – JMS Listener. JMS Listener manages the task of retrieving messages from a JMS destination (queue). This significantly reduces the development effort for the JMS-enabled applications.

The implementation of JMS Listener within WebSphere Application Server Enterprise Edition is very close to the current level of the Message-Driven Bean (MDB) specification that is part of the Enterprise JavaBean version 2.0 (EJB 2.0) specification being ratified by Sun.

JMS Listener can be considered as an input mechanism to access business logic that already exists. A client application places a message on a JMS destination (queue). JMS Listener detects the message and invokes a message bean according to the listener's XML configuration file. The work is then delegated to other components of the system, such as Enterprise JavaBeans (EJBs) or JavaBeans. In this scenario, business logic is separated from message handling.

Two models currently exist for JMS:

- **JMS point-to-point**: Messages arrive at a JMS destination, a physical queue. They are then routed to an EJB called the *message bean*. The message bean performs processing that can be based on the content of the message. This is illustrated in Figure 4-1.

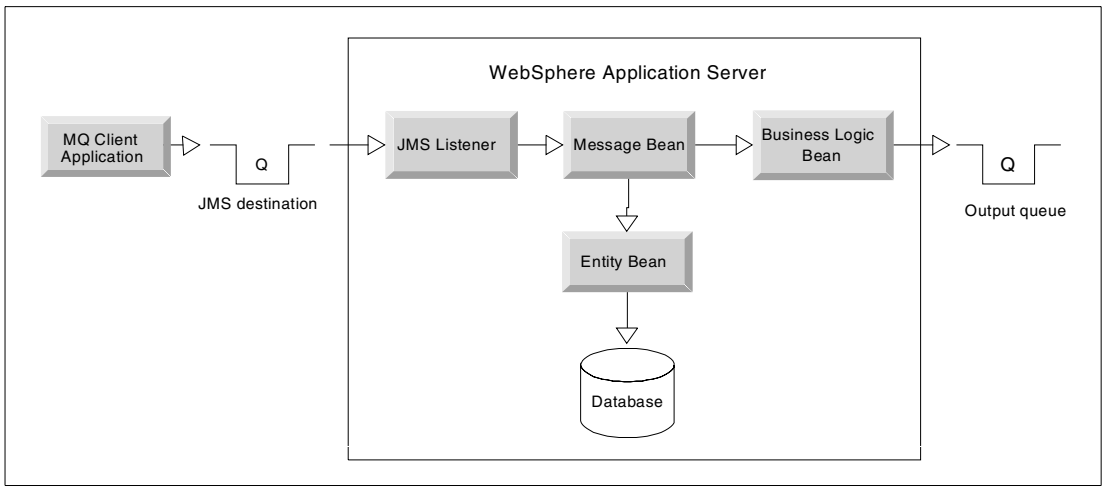


Figure 4-1 The JMS point-to-point model

Note: For completeness of the picture, we added an output queue, although it's not required by the JMS specification. JMS Listener provides support *only* for an input queue. In the chapter, we use term *JMS destination* for an input queue.

- **JMS publish/subscribe**: Messages are *published* to a JMS destination, called a *topic*. They are then distributed to all message beans that *subscribed* to that topic, via the MQSeries message broker and JMS Listener. Further processing of a message is similar to the point-to-point model (see Figure 4-2).

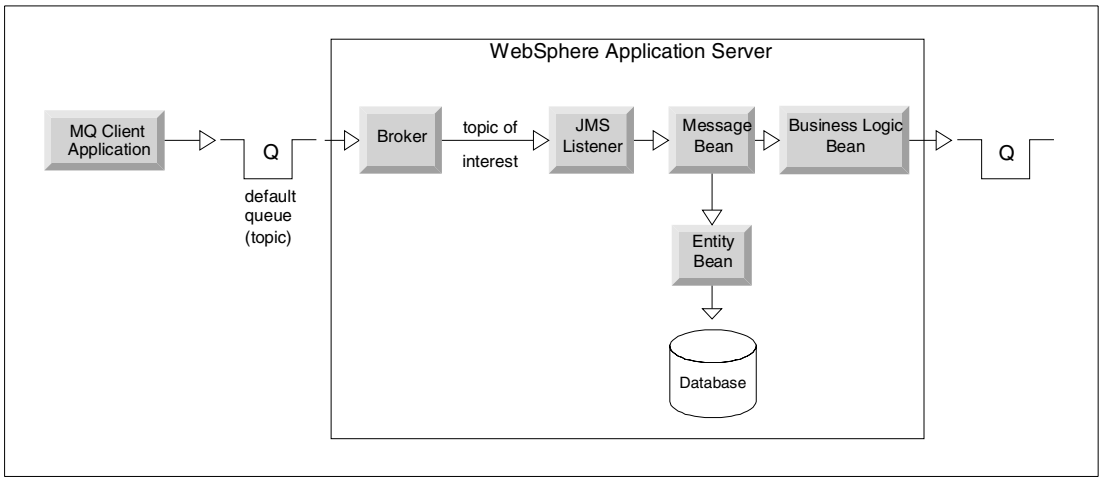


Figure 4-2 The JMS publish/subscribe model

Note: Walk-through point-to-point and publish/subscribe samples are also installed with WebSphere Application Server 4.0. You can find instructions in the %WAS_HOME%\Enterprise\samples\messaging\doc\PtoP and the %WAS_HOME%\Enterprise\samples\messaging\doc\PubSub directories.

4.2 Configuring JMS point-to-point for the OrderEntry application

A new function in the OrderEntry application offers you the ability to receive an inventory update notification from an external stock control system. We select the point-to-point JMS messaging model. To add this functionality, we need to configure the JMS Listener service and develop a message bean. We call it the `StockDeliveryUpdate` message bean. This bean performs the following steps:

1. Perform a JNDI lookup of the `Stock` entity bean, representing the stock level.
2. Update the inventory level for a given item at a given warehouse, by a given quantity.

The information to the `StockDeliveryUpdate` message bean is delivered as a comma-separated list of values within the body of a JMS message. Finally, confirmation of the message process is placed on an output queue. This is illustrated in Figure 4-3.

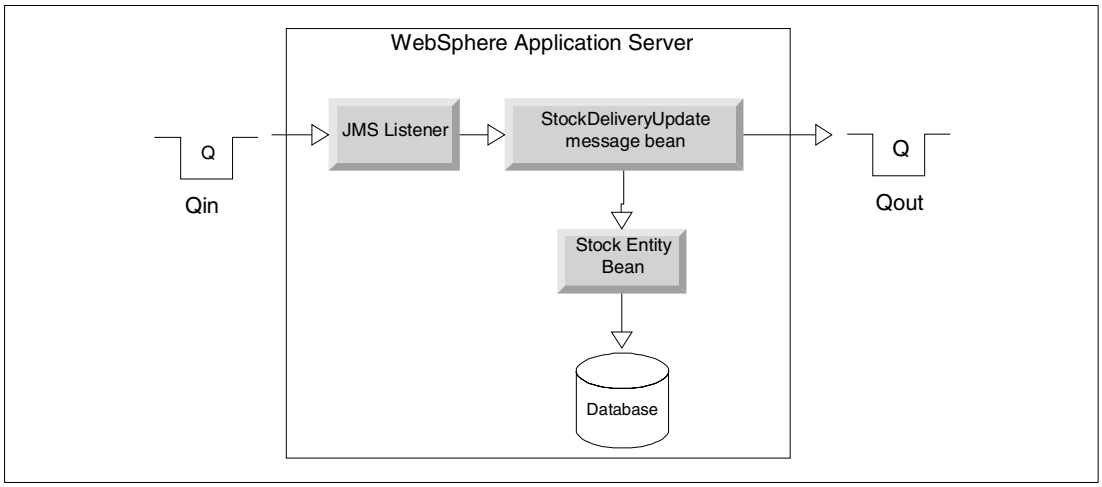


Figure 4-3 JMS point-to-point as applied to the OrderEntry application

The four main components of the JMS Listener service are configured before we can deploy the message bean. Let's discuss this configuration.

4.2.1 The core components of JMS point-to-point

Several core components enable the JMS Listener service for Java applications within WebSphere Application Server. We have to configure each component shown in Figure 4-4.

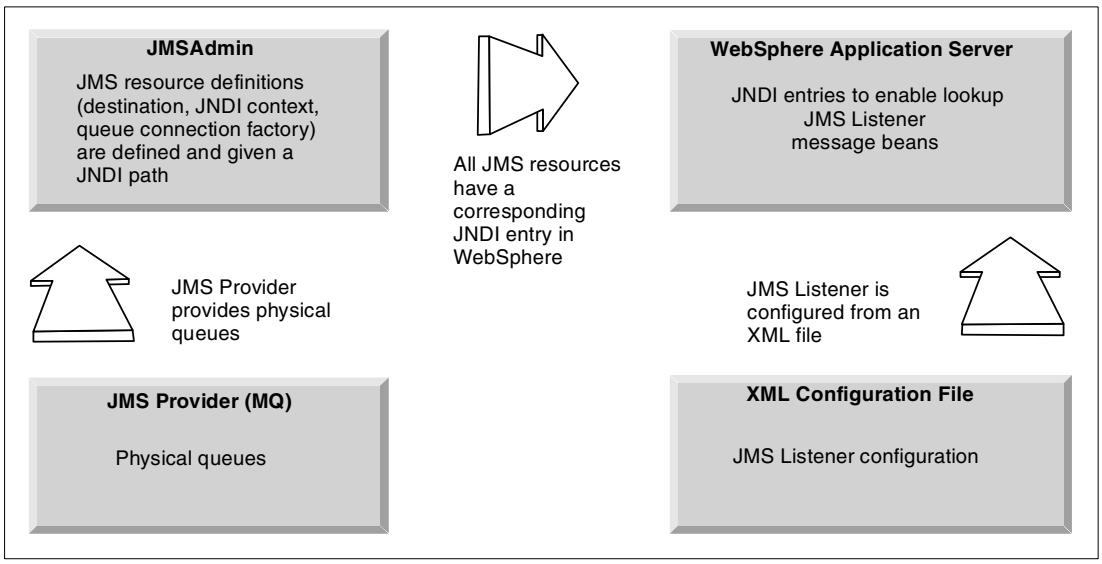

JMSAdmin
JMS resource definitions (destination, JNDI context, queue connection factory) are defined and given a JNDI path
All JMS resources have a corresponding JNDI entry in WebSphere
WebSphere Application Server
JNDI entries to enable lookup
JMS Listener
message beans
JMS Provider provides physical queues
JMS Listener is configured from an XML file
JMS Provider (MQ)
Physical queues
XML Configuration File
JMS Listener configuration

Figure 4-4 The core components of JMS in the point-to-point model

The four components are split between WebSphere Application Server Enterprise Edition and IBM MQSeries (JMS Provider). This defines a natural approach in the JMS Listener configuration process, which we explore in the following sections:

- On the JMS Provider side
  - Provide physical queues
  - Define JMS resources and assign them a JNDI path to enable them to be found via JNDI lookup. The JMSAdmin tool is supplied by MQSeries. The following three steps need to be performed:

i. Create JNDI context, the context definitions within which the JMS resources reside.

ii. Bind JNDI names to the physical queues (other queues, beside JMS destination, may be used by the message bean to send information to other systems).

iii. Specify the queue connection factory, the Java interface that is used to create a connection from the Java application to the JMS destination.

- On the WebSphere Application Server Enterprise Edition side

  WebSphere Application Server, as the JNDI directory server, contains the JMS Listener service and the message bean-enabled OrderEntry application. To enable JMS Listener on the WebSphere Application Server side, perform these steps:

  a. Configure base JMS support

  b. Configure the JMS Listener support by modifying the XML configuration file. It contains the listener stanza, which links JMS destinations to listeners and listeners to message beans.

4.2.2 Defining physical queues

The JMS Provider, in our case IBM MQSeries, provides the physical queues used in the point-to-point model.

MQSeries Explorer is used for creating queues. Two queues are created within the MQSeries Explorer, by right-clicking the local machine node and then choosing **New-> Local Queue** (see Figure 4-5). We name them:

- **Qin**: This is the JMS destination for incoming messages (see Figure 4-3 on page 54)
- **Qout**: This queue is used to send a confirmation message back

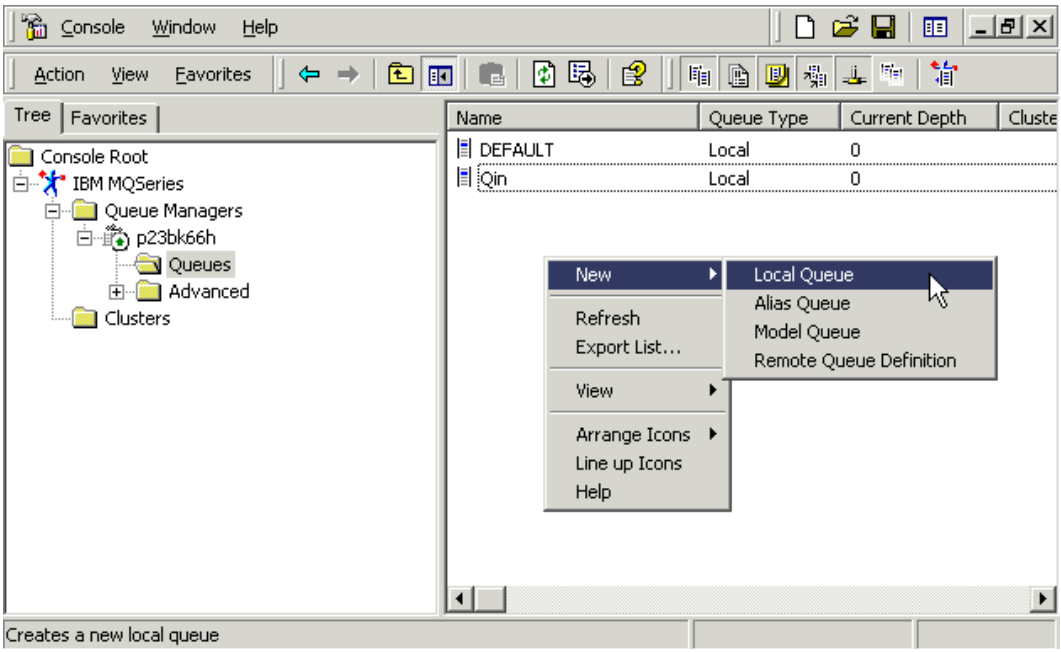


Figure 4-5 Defining and creating physical queues within MQ Explorer, the JMS Provider

4.2.3 Defining JMS resources

A key part of JMS implementation is the administration of JMS resources. This is performed within the JMSAdmin tool. This section discusses its configuration.

Figure 4-6 shows the relationship between the components in the JMS Provider (MQSeries) and WebSphere Application Server. All JMS resources are created and given a JNDI path by the JMSAdmin tool. The JMS destinations, which are defined in the tool, become the physical queues, provided by MQSeries. For each JMS resource, there is a corresponding JNDI entry defined in WebSphere Application Server.

Note: The JMS resource objects are bound into the namespace by the ResourceBinder object when WebSphere Application Server starts. After the server is started, the entire namespace can be viewed by using the DumpNameSpace tool shipped with WebSphere Application Server.

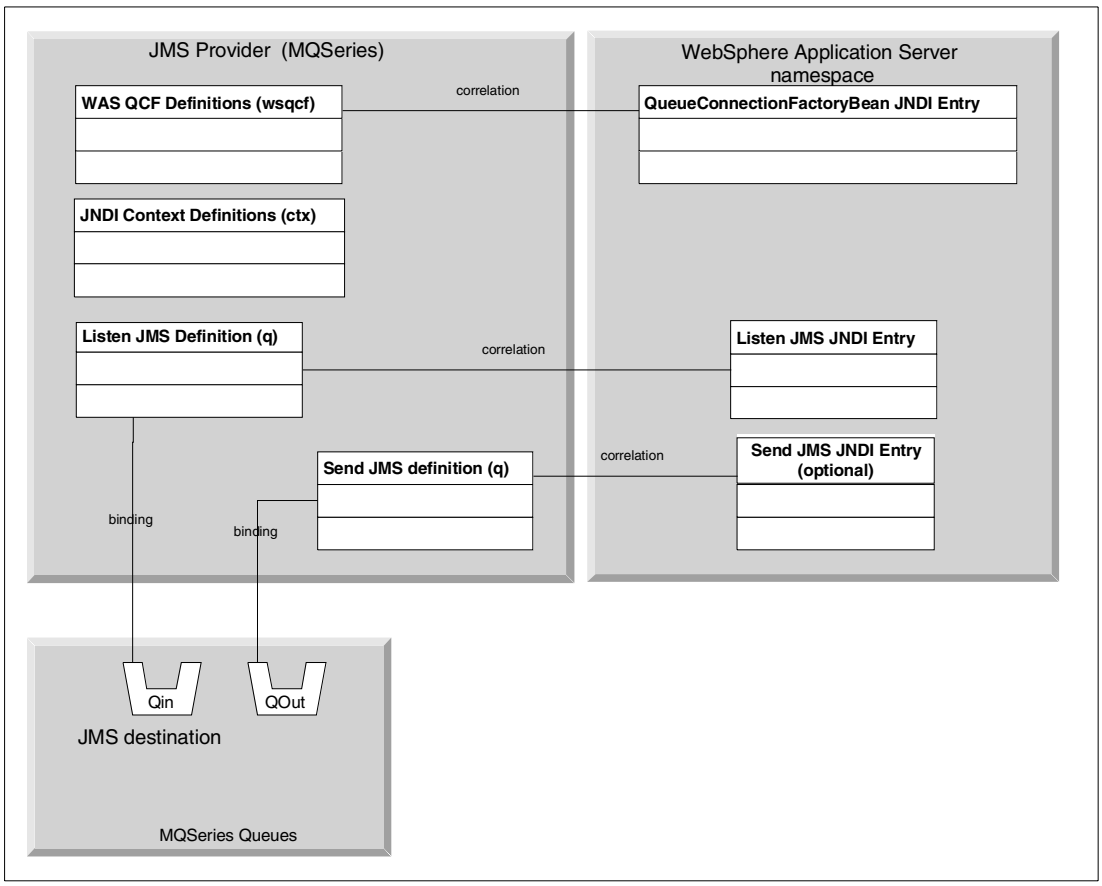


Figure 4-6 The JMS Provider and admin within the JMS architecture

The JMSAdmin tool is supplied with MQSeries. It can be found in the <MQSeries install home>\java\bin directory.

Edit the JMSAdmin.bat file as follows to ensure that it uses the version of Java provided by WebSphere and includes additional directories in its run time:

1. Add the following path onto the Java call, within the JMSAdmin.bat file:

```
%WAS_HOME%/java/jre/bin
```

2. Provide the following directive to the Java run time as an option:

```
-Djava.ext.dirs=%WAS_HOME%\java\lib;
                %WAS_HOME%\java\jre\lib\ext;
                %WAS_HOME%\lib;
                %WAS_HOME%\java\lib
                %MQSeriesHome%\Java\lib
```

Here *%WAS_HOME%* is the environment variable that points to the installation directory for WebSphere Application Server. *%MQSeriesHome%* is an environment variable pointing to the installation home directory of MQSeries. Refer to the sample JMSAdmin.bat file in Example 4-1.

Example 4-1 JMSAdmin script used for the OrderEntry sample

```
cls
@echo off

rem ---------------------------------------------
rem  IBM MQSeries JMSAdmin Tool Execution Script
rem  for Windows NT
rem
rem  Note that the properties passed to the java
rem  program are defaults, and should be edited
rem  to suit your installation if necessary
rem ---------------------------------------------

set WAS_HOME=C:\IBM\WebSphere\AppServer
set MQSeriesHome=C:\IBM\MQSeries

set libPaths=;
set libPaths=%libPaths%;%WAS_HOME%\java\lib;
set libPaths=%libPaths%;%WAS_HOME%\java\jre\lib\ext;
set libPaths=%libPaths%;%WAS_HOME%\lib;
set libPaths=%libPaths%;%MQSeriesHome%\java\lib

java  -Djava.ext.dirs="%libpaths%" -DMQJMS_LOG_DIR="%MQ_JAVA_INSTALL_PATH%"\log
-DMQJMS_TRACE_DIR="%MQ_JAVA_INSTALL_PATH%"\trace
-DMQJMS_INSTALL_PATH="%MQ_JAVA_INSTALL_PATH%" com.ibm.mq.jms.admin.JMSAdmin %1 %2 %3 %4 %5
```

The JMSAdmin tool is invoked at the command line by entering the **jmsadmin** command at a command prompt, within the <MQSeries install directory>\Java\bin directory.

In doing so, the tool places you at the root context for JMS, called `InitCtx` as shown in Figure 4-7.

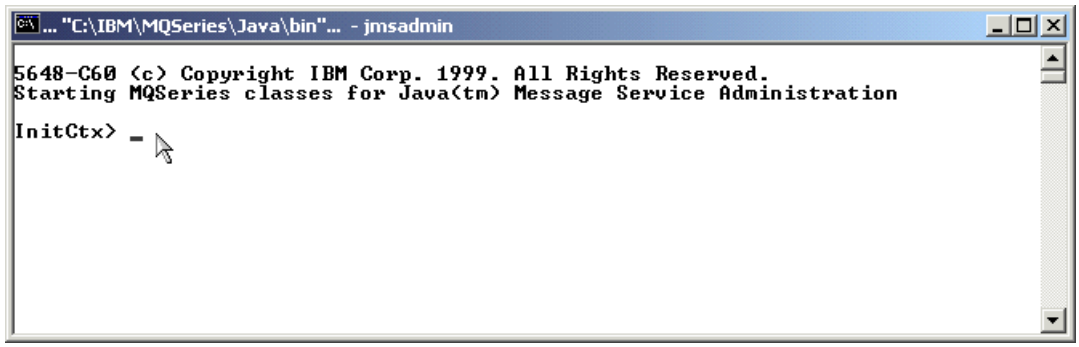

... "C:\IBM\MQSeries\Java\bin"... - jmsadmin
5648-C60 (c) Copyright IBM Corp. 1999. All Rights Reserved.
Starting MQSeries classes for Java(tm) Message Service Administration
InitCtx>

Figure 4-7 JMSAdmin places you at the root context when it launches

Note: You can find details of JMS resources, defined and administered by the JMSAdmin tool, in Section 4.6.3.4, "Support for the use of MQSeries Java Message Service resources" on the Web at:
http://www.ibm.com/software/webservers/appserv/doc/v40/aee/index.html

JNDI context definitions

We chose /OrderEntry/JMS as the top level context for our JMS resources. Each of the resources in Figure 4-6 are defined using the following administration commands:

```
Def ctx(OrderEntry)
chg ctx(OrderEntry)
Def ctx(JMS)
chg ctx(JMS)
```

The **Def** command *defines* a JMS context. The **chg** command *changes into* the JMS context.

To display the contents of your current context, enter the following command:

```
dis ctx
```

Output from the command is shown in Figure 4-8.

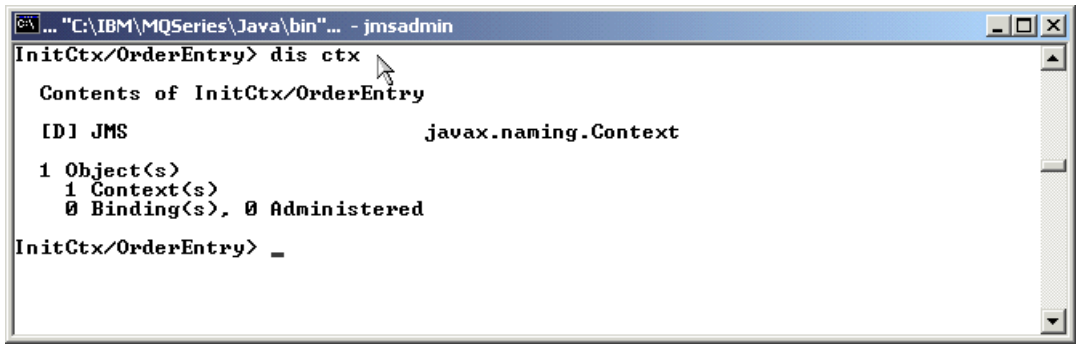


Figure 4-8 The current context

To delete the context, you must be within its parent context and issue the following command:

```
Del ctx(<context name>)
```

The JMS destination and other queues

The next step is to add information in the namespace (under /OrderEntry/JMS context) that binds JNDI names into the physical queues.

The JMS destination, *Qin*, is defined and bound to a JNDI entry using the JMSAdmin tool. The same operation is performed for the output queue, *Qout*.

While within the /OrderEntry/JMS JNDI context, we execute the following commands:

```
Def q(Qin) Qu(Qin)
Def q(Qout) Qu(Qout)
```

q is the JMS queue resource, and *Qu* is the physical MQSeries queue.

A list of the newly defined JMS resources at the /OrderEntry/JMS node in the JNDI namespace can be displayed by entering the command:

```
dis ctx
```

The output of this command is shown in Figure 4-9.

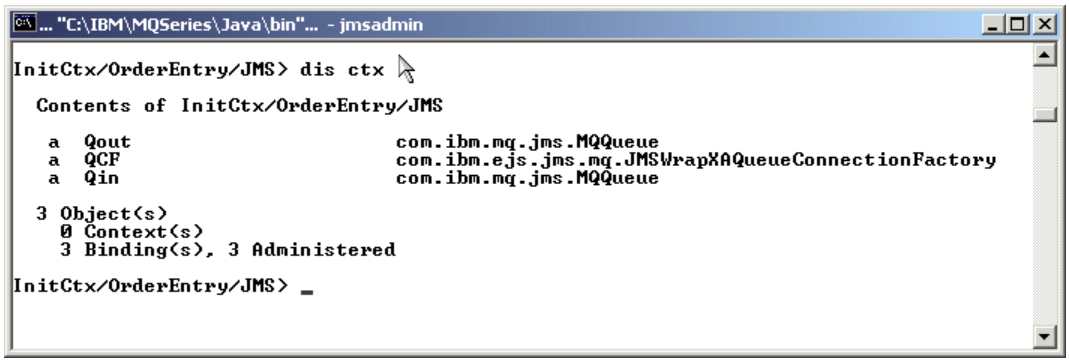


Figure 4-9 The output from the dis ctx command in the JMSAdmin tool

You can delete queue definitions by entering the following command and passing the name of the queue to be deleted as a parameter:

```
Del q(Qin)
```

The queue connection factory

The queue connection factory is an interface used to create queue objects for use within Java applications. A JMS connection is a client's active connection to its JMS Provider queue.

We have to define the queue connection factory too. While within the /OrderEntry/JMS JNDI context, we create the queue connection factory by entering the following command:

```
Def WSQCF(QCF)
```

This creates the WebSphere connection factory including default parameters, as seen in Figure 4-9. WebSphere queue connection factories support transactions in JMS message beans. You can find further information about this in 4.4.1, “Transaction support” on page 84.

The JMSAdmin tool supports a number of different queue connection factories, two of which are relevant to WebSphere Application Server:

- **Standard, non-transactional connection factory**: Defined in the JMSAdmin tool as QCF.
- **Transactional connection factory**: Defined as WSQCF in the JMSAdmin tool; supports transactional messaging, where messages can be placed back on the queue in the event of message bean processing failure.

4.2.4 WebSphere Application Server

The naming server within WebSphere Application Server is used to locate all JMS resources via JNDI. It also manages the JMS Listener and message beans.

The JMS Listener service is hosted by WebSphere Application Server. At run time, it uses the XML configuration file to create listeners. Each listener invokes a specified message bean when a message arrives at an associated JMS destination queue. JNDI entries are used to locate all JMS resources. These entries must be entered into WebSphere Application Server to enable JNDI lookup of the resources.

Figure 4-10 shows WebSphere Application Server in relation to the JMS Provider and the XML configuration file. The XML tags found in the file relate to JMS resources, whose entries are found in the JNDI namespace managed by WebSphere Application Server. The JNDI name of the message bean is also managed by WebSphere Application Server.

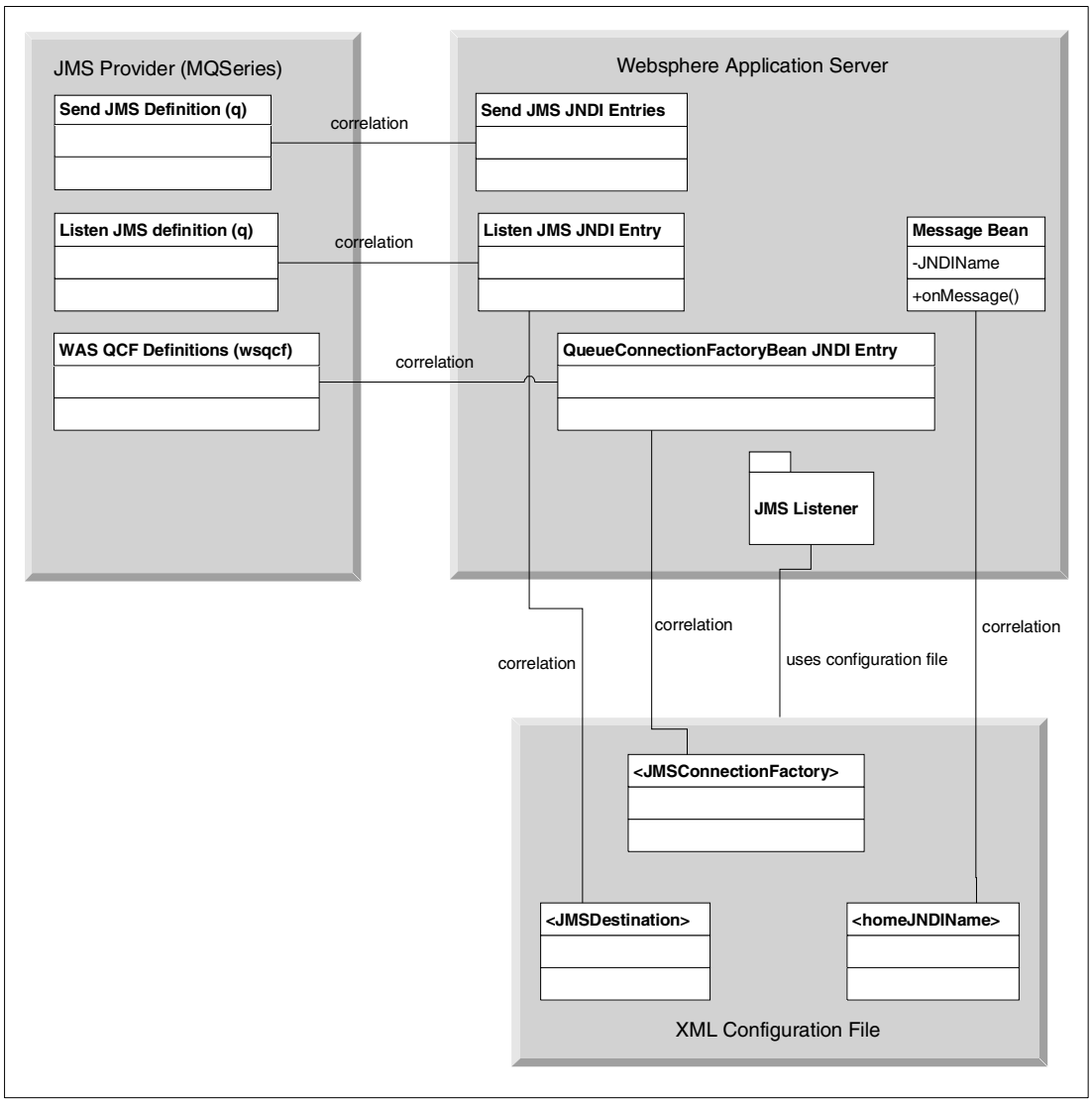


Figure 4-10 WebSphere Application Server and its relationships in the JMS architecture

JMS Provider

We configure the JMS Provider within WebSphere Application Server by using these steps:

1. Expand **Resources-> JMS Providers-> IBM MQSeries**.
2. Select **IBM MQSeries**.
3. Click the **Nodes** tab.
4. Click **Install New**.
5. Select the local node.
6. Click **Specify Driver**.
7. Click **Add Driver**.
8. Select **<MQSeries install home>\java\lib\fscontext.jar**. Repeat this step for providerutil.jar (see Figure 4-11).

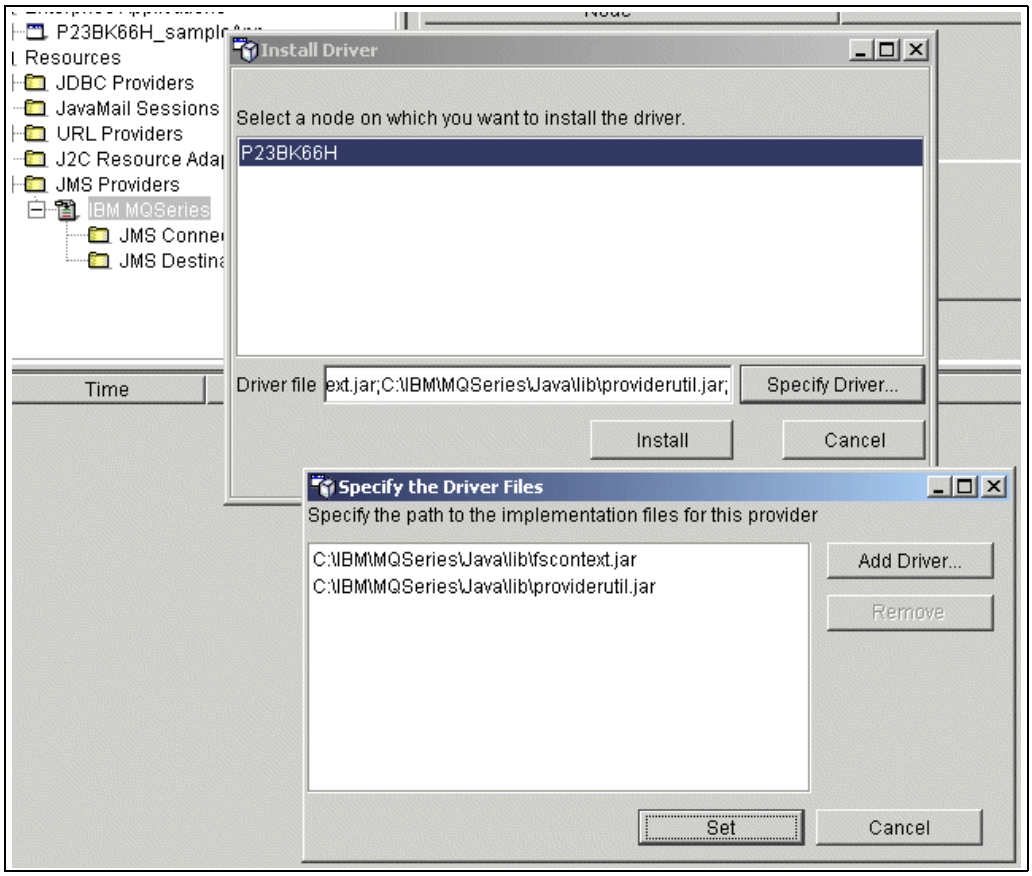


Figure 4-11 Configuring the JMS Provider

JNDI entries

For each resource declared using the JMSAdmin tool, we define a corresponding JNDI entry within WebSphere Application Server.

Figure 4-12 shows the JMS destination (queue called *Qin*) and an output queue, *Qout*. We define these JNDI entries by using these steps:

1. Expand the **Resources-> JMS Providers-> MQSeries** nodes within the administrator console.
2. Right-click **JMS Destinations**.
3. Select **New**.
4. Specify the JMS destination **Name** and its **External JNDI Path**.

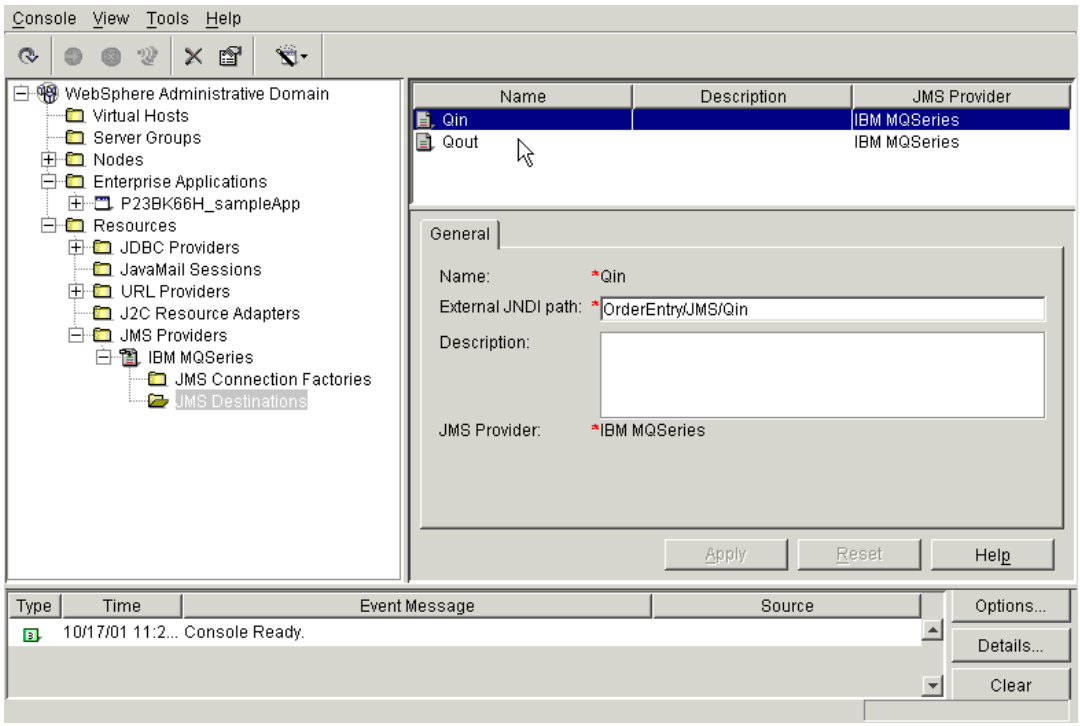


Figure 4-12 WebSphere Application Server queue JNDI entries

Figure 4-13 shows the queue connection factory definitions for our OrderEntry application. To add it, follow these steps:

1. Right-click **JMS Connection Factories**.
2. Select **New**.
3. Specify the JMS connection factory Name and its External JNDI Path.

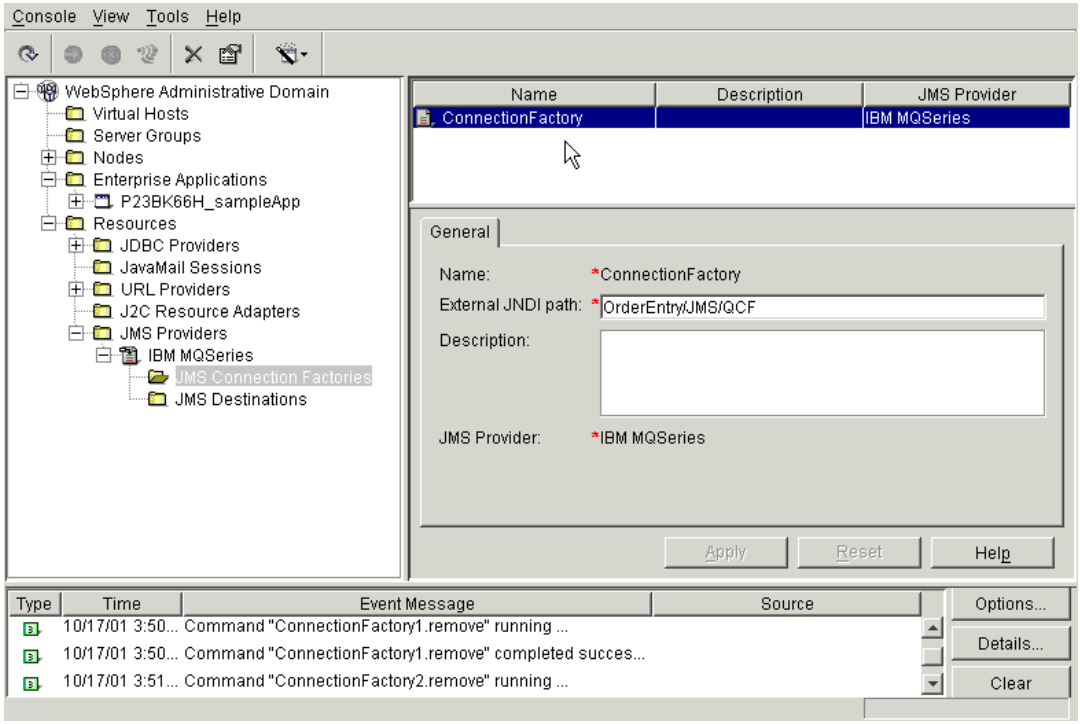


Figure 4-13 WebSphere Application Server queue connection factory JNDI entries

The queues and the queue connection factory can now be found through a JNDI lookup against WebSphere Application Server.

JMS Listener

JMS Listener is installed with the JMS Enterprise Service. Its functionality is to route messages arriving at a JMS destination to a message bean for processing, as illustrated in Figure 4-14. JMS Listener is supplied as part of WebSphere Application Server Enterprise Edition. It runs within WebSphere Application Server.

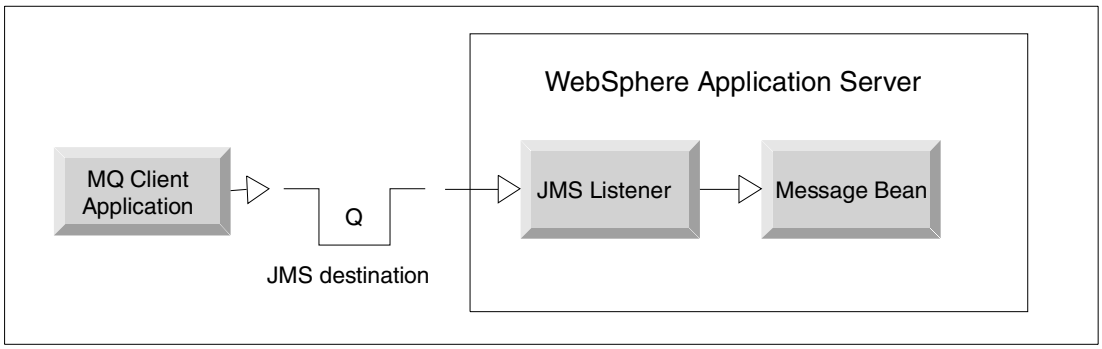


Figure 4-14 JMS Listener routes messages to a message EJB

The work of managing JMS Listeners is done by JMS Listener manager. It provides the following services:

- Creates a pool of dynamic session threads for use by JMS Listeners
- Creates and starts JMS Listeners based on the XML configuration file (the XML configuration file is covered in 4.2.5, “JMS Listener XML configuration file” on page 65)
- Controls the cleanup of resources during server termination

Each JMS Listener completes several steps for the JMS destination that it is to monitor, including:

- Creating a JMS server session pool and allocating JMS server sessions and session threads for incoming messages
- Creating JMS connection consumers to listen for incoming messages
- If specified, starting a transaction and requesting that it is committed (or rolled back) when a message bean completes its processing
- Processing incoming messages by invoking the *onMessage()* method of the specified message bean

To enable JMS Listener, we have chosen the existing default Application Server in the topology. (However, you can use any existing server to perform this exercise.) Then, ensure it has the Extended Messaging Service enabled (see Figure 4-15):

1. Expanded **Nodes -> *<local machine node>*-> Application Servers-> Default Server**.
2. Click the **Custom** tab in the server configuration pane.
3. Select **Extended Messaging Support**.
4. Click **Edit**.

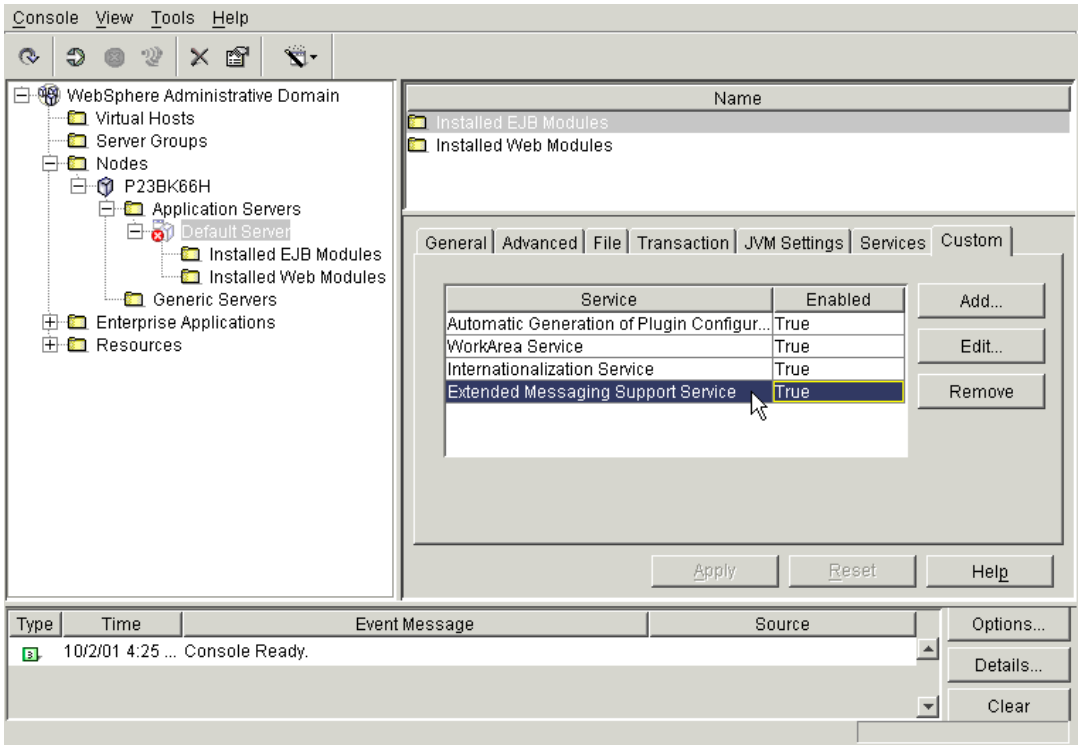


Figure 4-15 Ensuring an Application Server has Extended Messaging enabled

Note: It is possible to create a new Application Server specifically for use with JMS. This enables setting properties on the server in isolation and specifically for the JMS Listener service. Keep in mind, JMS Listener will run on its own JVM. This means that calls to EJBs on other Application Servers will become “out of process”. You must also consider scaling issues with JMS Listener:

- How many destinations should it be listening upon?
- How many bean instances should be active?
- How big should the pool be defined for sessions?
- What happens if you define two servers on the same machine, both with a listener active. Should they share the same destination or interference a problem?

5. The final part of the JMS Listener configuration is shown in Figure 4-16. We ensure:
   – The classpath for the JMS Listener contains the mq.jar and jms.jar files, supplied with MQSeries.
   – The classname for the class that implements the JMS Listener itself is specified.
   – The XML configuration file that we wanted to use is specified.

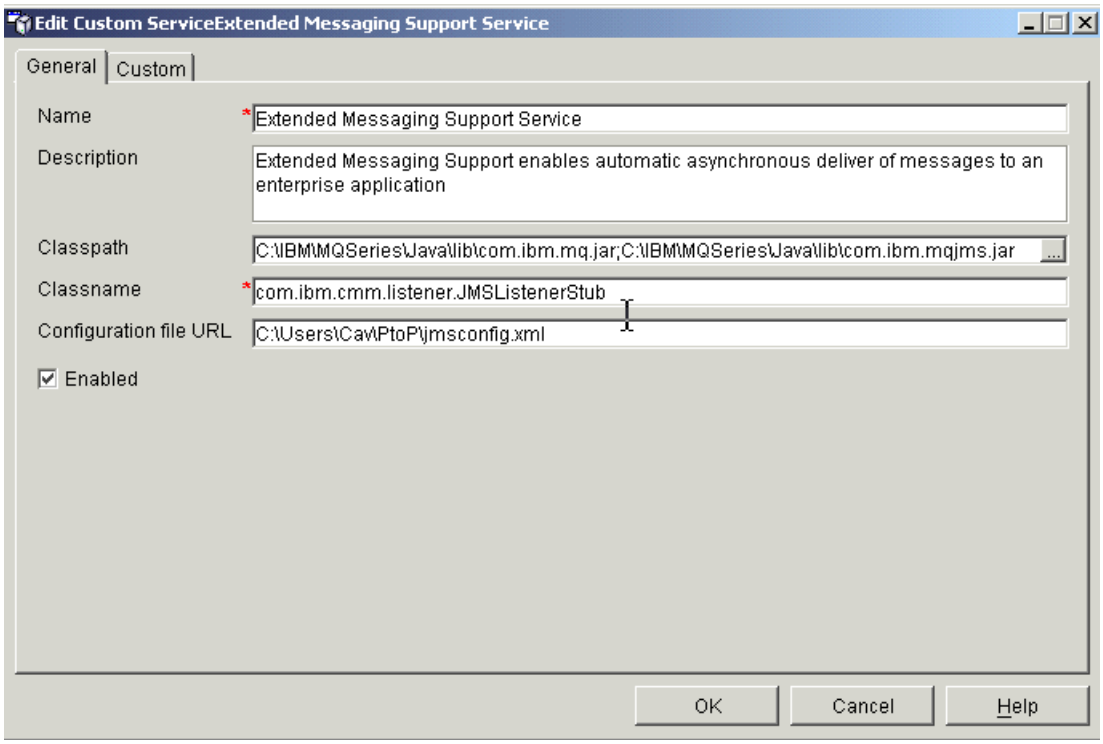


Figure 4-16 Configuring the JMS Listener service for the point-to-point model

4.2.5 JMS Listener XML configuration file

The operation of the Extended Messaging Service is controlled by an XML configuration file, which you edit to specify properties of the service. Consider, for example, the identities of JMS destinations that are to be monitored, the transactional requirements, and the name of the message bean to process messages. When an application server is started, it initializes the JMS Listener manager based on the JMS Listener XML configuration file. For each message bean that takes part in JMS, there is a corresponding listener entry in the JMS XML configuration file.

Each JMS Listener configuration is defined within the <*Listener*> XML tag of the XML configuration file. It is sometimes referred to as the *listener stanza*.

Figure 4-17 shows the relationship between JNDI entries in WebSphere Application Server and XML tags in the listener stanza of the configuration file. *homeJNDIName* for the message bean is registered with JNDI when the message bean is installed in WebSphere Application Server.

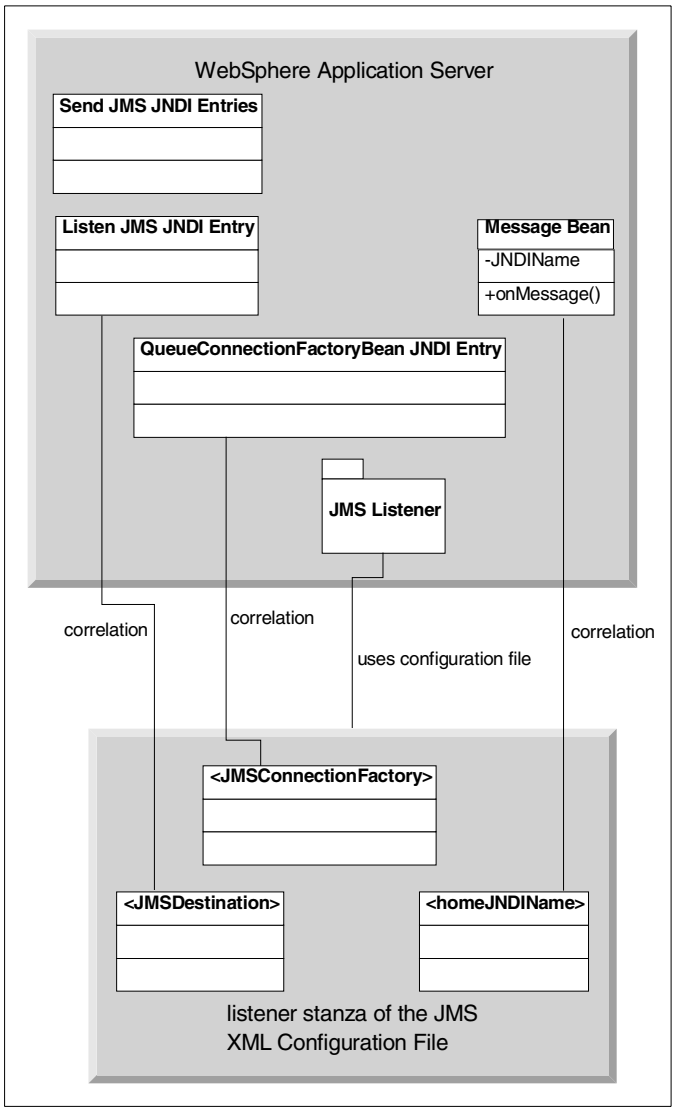


Figure 4-17 The XML configuration file and its relationship to other components

The XML configuration file we used for our example is shown in Example 4-2. The name of the message bean is `StockDeliveryUpdate`.

Example 4-2 XML configuration file used in our OrderEntry application

```
<Config>
<Pooling>
 <Timeout>100636</Timeout>
 <Threshold>36</Threshold>
</Pooling>
```

```
<Listener>
 <HomeJNDIName>OrderEntry/JMS/StockDeliveryUpdate</HomeJNDIName>
 <JMSConnectionFactory>OrderEntry/JMS/QCF</JMSConnectionFactory>
 <JMSDestination>OrderEntry/JMS/Qin</JMSDestination>
 <MaxRetries>6</MaxRetries>
 <MaxSessions>1</MaxSessions>
</Listener>
</Config>
```

4.2.6 The message bean

When a message arrives at the JMS destination, JMS Listener gets this message and passes it to the message bean that has been specified in the JMS Listener configuration file. The message bean is responsible for processing the message. It is good practice when programming a message bean to delegate business functions to another EJB. This provides a clear separation of message handling and business processing. The message bean may perform processing within the scope of a transaction.

The current implementation of a message bean uses a stateless session EJB. IBM's future direction with regard to messaging is to enable support for Message-Driven Beans (MDBs), as defined within the EJB 2.0 specification.

To be treated as a message bean, your EJB must have an *onMessage()* method, which should maintain the following rules:

- The method must be declared as public.
- The method must not be declared as final or static.
- The return type must be void.
- The method must have a single argument of type `javax.jms.Message`.
- The throws clause must not define any application exceptions:

```
public void onMessage(javax.jms.Message incomingJMSMessage)
{
.....
}
```

It is also worth considering the function provided in the *ejbCreate()* and *ejbRemove()* methods. For these methods, we recommend that they follow the Message-Driven Beans requirements:

- The method must be declared as `public`.
- The method must not be declared as final or `static`.
- The return type must be `void`.
- The method must not have any arguments.
- The throws clause must not define any application exceptions.

In addition, for your message beans, you should avoid standard session programming practices that will inhibit the migration of message beans to MDBs, specifically:

- **Do not write any standard EJB client code**.

  You should never look up the JMS Listener's home interface in any of your application code or obtain a handle for the home interface. MDBs do not have a home interface.

- **Do not implement any *create(...)* methods on the home interface other than the create method, with no arguments.**

  This is typically not done on stateless session beans but is also not explicitly prohibited by the EJB 1.1 specification.

- **Do not implement any additional business methods (beyond the *onMessage()* method) in the message bean's remote interface.**

  MDBs do not have a remote interface. Note that it is okay to have additional supporting methods on the bean itself; just do not count on them being callable anywhere outside of the listener itself. To this end, you should consider making all supporting methods private.

- **Do not write any code in a message bean that uses the optional *SessionSynchronization* interface.**

  This interface is not supported by MDBs.

- **Isolate your use of *getSessionContext()* calls appropriately.**

  MDBs will have a *getMessageDrivenContext()* method instead that will return a `javax.ejb.MessageDrivenContext` object.

We developed a message bean, `StockDeliveryUpdate`, that processes all messages from the *Qin* JMS destination. As a message is passed to the `StockDeliveryUpdate` bean, it accesses the `Stock` entity bean to increase the inventory quantity for a particular product at a particular warehouse.

Note: The message bean uses the Java classes supplied by MQSeries. The MQ classes for Java are available from the MQSeries Web site as a support pack: `http://www.ibm.com/software/ts/mqseries/txppacs/ma88.html`

The bean is developed using WebSphere Studio Application Developer (see Figure 4-18):

1. Use the EJB creation wizard. The code produced by the wizard is compatible with the requirements stated above. The bean creation dialogue is invoked by expanding the **EJB Modules** tree.
2. Right-click **OrderEntry EJBs** in the J2EE view.
3. Select **New-> Enterprise Bean**.

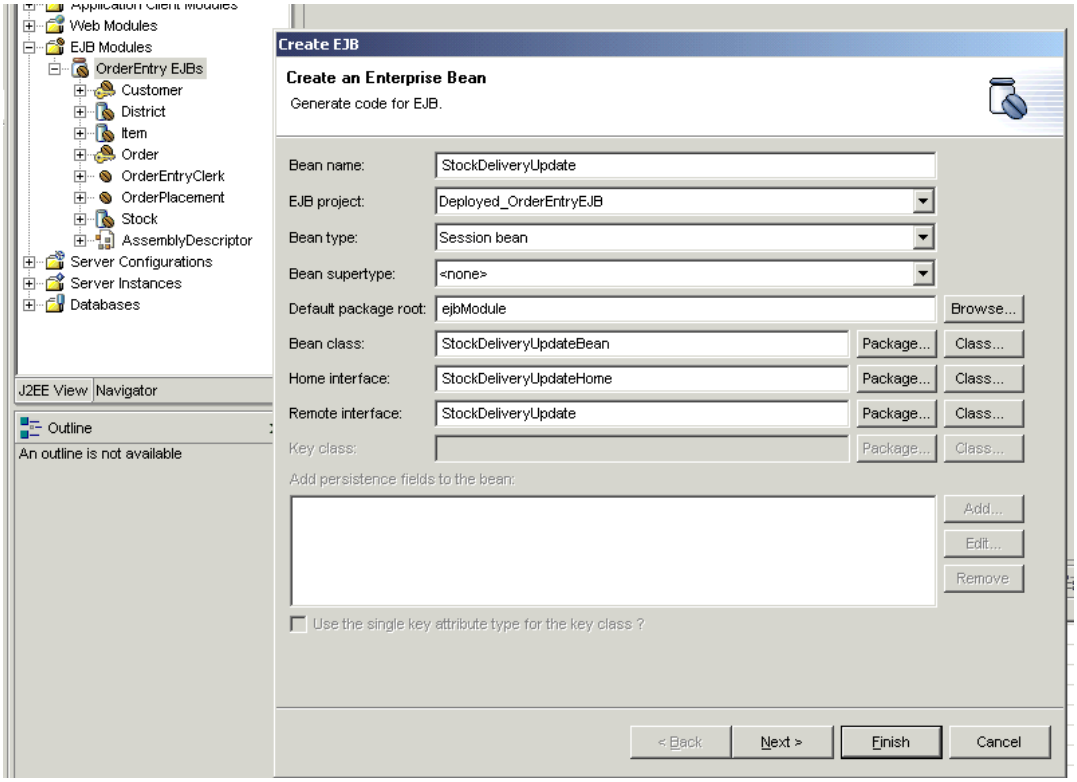


Figure 4-18 Creating the StockDeliveryUpdate bean for the OrderEntry application

4. Add the *onMessage()* method (shown in Example 4-3). This is a required method.

Example 4-3 The onMessage() method

```
public void onMessage(javax.jms.Message inputMessage) {
   System.out.println("StockDeliveryUpdateBean.onMessage()...");
   try {

      StringTokenizer st =
         new StringTokenizer(((TextMessage) inputMessage).getText(), ",");
      System.out.println(
         "StockDeliveryUpdateBean.onMessage() ((TextMessage)inputMessage).getText():"
            + ((TextMessage) inputMessage).getText());

      int count = 0;
      String[] tempValues = new String[st.countTokens()];
      while (st.hasMoreTokens()) {
         tempValues[count++] = st.nextToken();
      }
      System.out.println(
         "StockDeliveryUpdateBean.onMessage() increasing stock count at warehouse:"
            + tempValues[0]
            + " for item:"
            + tempValues[1]
            + " by "
            + tempValues[2]
            + " items");
      int iStockLevel =
```

```
            updateStockValue(tempValues[0], tempValues[1],
Integer.parseInt(tempValues[2]));

         System.out.println(
            "StockDeliveryUpdateBean...placing confirmation message on output queue...");
         putMessage(
            "OrderEntry/JMS/Qout",
            null,
            "onMessage succeeded. new stock level is: " + iStockLevel);
         System.out.println(
            "StockDeliveryUpdateBean...placing confirmation message on output
queue...end.");
      }
      catch (Exception oException) {
         System.out.println("StockDeliveryUpdateBean...exception...");
         oException.printStackTrace();
         System.out.println("StockDeliveryUpdateBean...exception...end.");

         System.out.println(
            "StockDeliveryUpdateBean...placing confirmation message on output queue...");
         try {
            putMessage("OrderEntry/JMS/Qout", null, "onMessage failed");
         }
         catch (Exception oException2) {
            System.out.println("StockDeliveryUpdateBean...exception...");
            oException.printStackTrace();
            System.out.println("StockDeliveryUpdateBean...exception...end.");
         }
         System.out.println(
            "StockDeliveryUpdateBean...placing confirmation message on output
queue...end.");
      }

      System.out.println("StockDeliveryUpdateBean.onMessage()...end.");
}
```

5. We also add several utility methods. The first of which is *getInitialContext()*. It is used to find the initial context for JNDI lookup of resources used by the bean. These resources are the `Stock` entity bean and the *Qout* queue. The method's listing is shown in Example 4-4. The method uses the resource bundle file to get information about the context factory, JNDI server name, and its port number.

Example 4-4 Method added to obtain the initial JNDI context

```
   private Context getInitialContext() throws Exception
   {
      ResourceBundle resources = ResourceBundle.getBundle("OrderResourceBundle");
      String JNDIcontextFactoryName = resources.getString("factory");
      String JNDIServerName = resources.getString("system");
      String JNDIServerport = resources.getString("port");

      Properties oProperties = new Properties();
      oProperties.put(Context.INITIAL_CONTEXT_FACTORY, JNDIcontextFactoryName);
      oProperties.put(Context.PROVIDER_URL, "iiop://" + JNDIServerName + ":" +
JNDIServerport);
```

```
      InitialContext oInitialContext = new InitialContext(oProperties);
      return oInitialContext;
}
```

6. Add the next method called *updateStockValue()* (shown in Example 4-5). It increases the stock level of an item at a warehouse. The *updateStockValue()* method invokes the *increaseStockQuantity()* method of the Stock bean in order to increase inventory.

Example 4-5 Method added to increase the stock level for an item at a given warehouse

```
   private int updateStockValue(String warehouseId, String itemId, int newStockDelivered)
throws Exception{
      // Get the home interface
      // Get references to EJB homes and the main EJB
      Context ctx = getInitialContext();

      // find the Stock entity bean, using the EJB reference
      java.lang.Object tempObject = ctx.lookup("java:comp/env/ejb/Stock");
      StockHome stockHome =(StockHome) javax.rmi.PortableRemoteObject.narrow(
                           tempObject,
                           StockHome.class);

      // Get the entity bean with a specific primary key
      com.ibm.itso.roch.wasaejb.Stock stockEJB = null;
      StockKey stockKey = new StockKey(warehouseId, itemId);
      try{
         System.out.println("StockDeliveryUpdateBean...finding stock bean...");
         stockEJB =
               (com.ibm.itso.roch.wasaejb.Stock) stockHome.findByPrimaryKey(stockKey);
         System.out.println("StockDeliveryUpdateBean...finding stock bean...end.");
      }catch (FinderException oException){
         System.out.println("StockDeliveryUpdateBean...exception...");
         oException.printStackTrace();
         System.out.println("StockDeliveryUpdateBean...exception...end.");
      }

      System.out.println("StockDeliveryUpdateBean...found bean, now updating values...");
      stockEJB.increaseStockQuantity(newStockDelivered);
      int iStockQuantity = stockEJB.getStockQuantity();
      System.out.println("StockDeliveryUpdateBean...stock value is now:" +
iStockQuantity);
      System.out.println("StockDeliveryUpdateBean...found bean, now updating
values...end.");

      return iStockQuantity;
   }
```

7. Create a method, called *putMessage()*, to place messages on an output queue, *Qout* (shown in Example 4-6). *Qout* was created using the MQSeries Explorer (refer to Figure 4-5 on page 55). The *putMessage()* method takes the target queue as a JNDI path, a message ID to enable message correlation, and the message data as a `String`.

   In Example 4-6, the key objects for posting messages to a JMS destination queue are declared at the beginning of the method body. They are:

   – `QueueConnectionFactory`: Used to create a queue connection
   – `QueueConnection`: The actual connection to the queue

- QueueSession: Manages the session from the time the connection is started, *oQueueConnection.start()*, to the time it is closed, *oQueueConnection.close()*
- QueueSender: Used to send the JMS message

Example 4-6 Placing a message at a JMS queue destination: The putMessage() method

```
   private void putMessage(String TargetQueueJNDIName, String msgID, String text) throws
Exception {
      QueueConnectionFactory oQueueConnectionFactory     = null;
      QueueConnection        oQueueConnection      = null;

      QueueSession           oQueueSession = null;
      QueueSender            oQueueSender  = null;

      System.out.println("StockDeliveryUpdateBean...request to put message... '" + text +
"'...");

      try
      {

        InitialContext oInitialContext = new InitialContext();

        //
        // Obtain the queue connection factory from JNDI
        //
        System.out.println("StockDeliveryUpdateBean...find QueueConnectionFactory...");
        oQueueConnectionFactory = (QueueConnectionFactory)
oInitialContext.lookup("OrderEntry/JMS/QCF");
        System.out.println("StockDeliveryUpdateBean...find
QueueConnectionFactory...end.");

        //
        // Create a queue connection
        //
        oQueueConnection = oQueueConnectionFactory.createQueueConnection();
        oQueueConnection.start();

        //
        // Create a session.
        //
        oQueueSession = oQueueConnection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);

        //
        // Obtain the Destination object from JNDI
        //
        System.out.println("StockDeliveryUpdateBean...find Queue...");
        Queue ioQueue = (Queue) oInitialContext.lookup(TargetQueueJNDIName);
        System.out.println("StockDeliveryUpdateBean...find Queue...end.");

        //
        // Create a QueueSender
        //
        System.out.println("StockDeliveryUpdateBean...creating queue sender...");
        oQueueSender = oQueueSession.createSender(ioQueue);
        System.out.println("StockDeliveryUpdateBean...creating queue sender...end.");

        //
        // Create a message to send to the queue...
```

```
        //
        TextMessage message = oQueueSession.createTextMessage(text);

        //
        // Set CorrelationID
        //
        if (msgID != null) {
         message.setJMSCorrelationID(msgID);
        }

        //
        // ...and send it
        //
        System.out.println("StockDeliveryUpdateBean...sending message to queue...");
        oQueueSender.send(message);
        System.out.println("StockDeliveryUpdateBean...sending message to queue...end.");

        //
        // Close the connection (close calls will cascade to other objects)
        //
        oQueueSender.close();
        oQueueSession.close();
        oQueueConnection.close();
        oQueueConnection = null;

        System.out.println("StockDeliveryUpdateBean...request to put message... '" + text
+ "'...end.");
      }
      catch (JMSException je)
      {
        System.out.println("putMessage failed with "+je);
        Exception le = je.getLinkedException();
        if (le != null) System.out.println("linked exception "+le);
      }
      catch (Exception e)
      {
        System.out.println("failed with "+e);
      }
      finally
      {
         oQueueConnection.close();
      }
   }
```

8. Generate the bean's deployed code. Right-click the bean in the J2EE view, and choose **Generate Deploy Code** (see Figure 4-19).

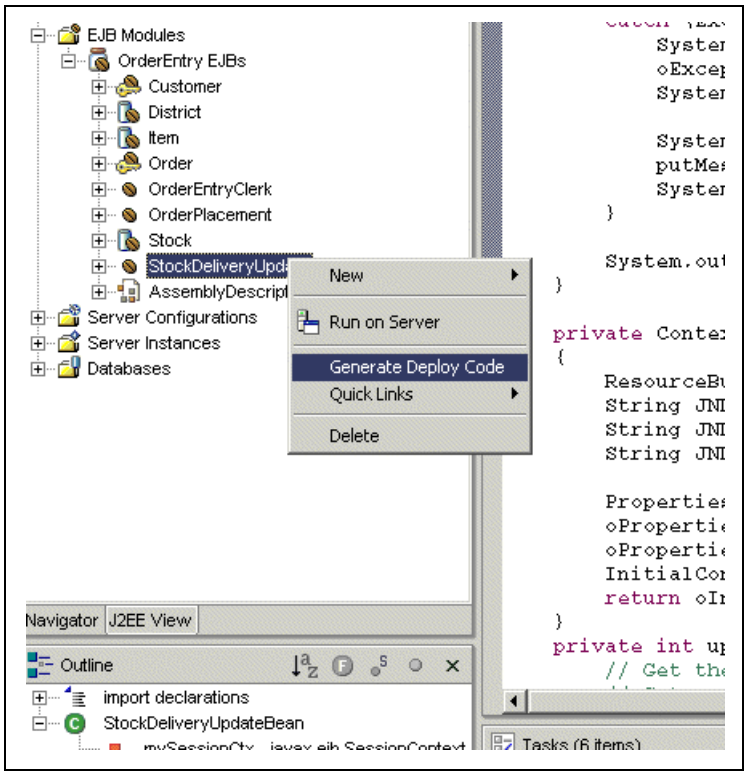


Figure 4-19 Generating EJB deployed code

9. Right-click **OrderEntry EJBs** in the J2EE view and select **Open With-> EJB Editor**.
10. Click the **References** tab.
11. Under **StockDeliveryUpdate**, add an EJB reference to the **Stock** entity bean and save the file.
12. The entire application is built by selecting **Project**, **Rebuild All**. This generates all the .class files from the source code for the message bean, including its stubs and RMI code. All Java source for your EJB project can be found by expanding your EJB project in the Navigator view.
13. Export the modified EAR file for the application and deploy it in WebSphere Application Server Enterprise Edition.

4.2.7 Testing the JMS sample code

To test the sample code, we need to send a message to the JMS destination, *Qin*. We use the MQSeries Explorer to perform this task (see Figure 4-20):

1. Start the application server that hosts the OrderEntry application.
2. Start MQSeries Explorer.
3. Expand the Queue Manager tree.
4. Expand the local machine node.
5. Select **Queues**.
6. Right-click the **Qin** queue, and select **Put Test Message**.
7. The message we place is a comma-separated list of integers: the factory ID, stock ID, and inventory level increase respectively, for example, `0001,000001,50`.

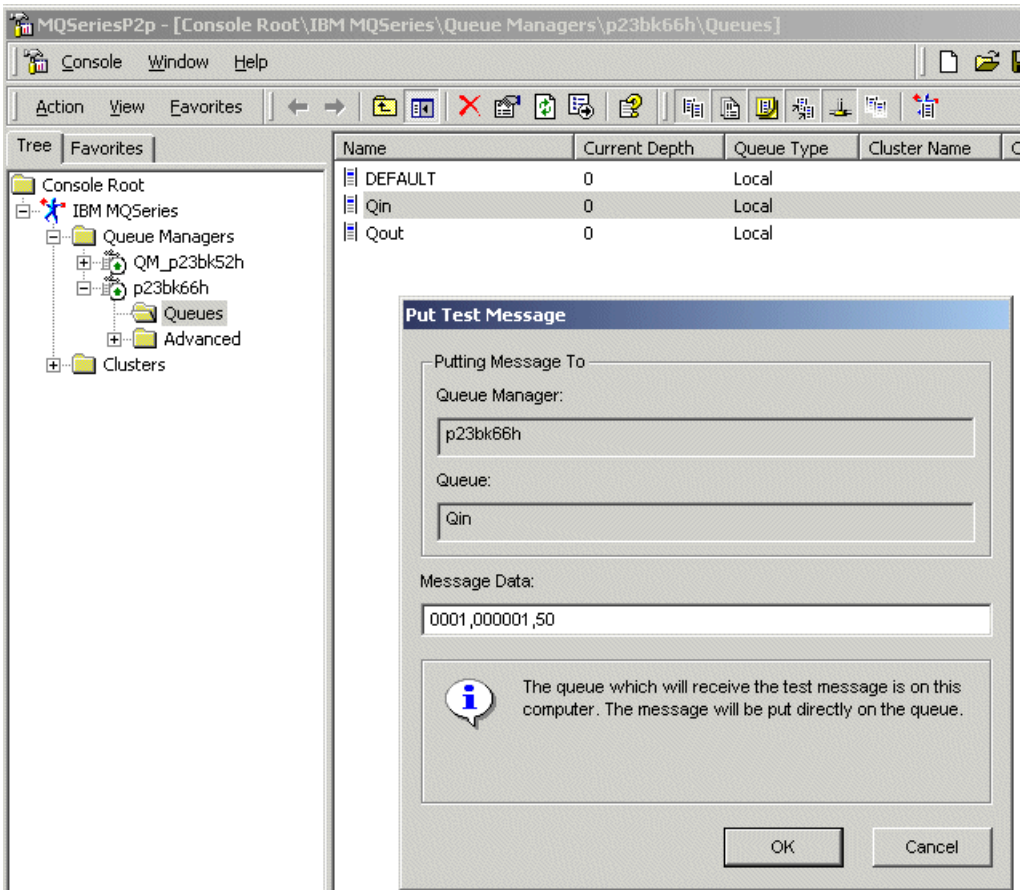


Figure 4-20 Injecting test messages into the Qin JMS destination queue

8. When the message bean, `StockDeliveryUpdate`, is invoked, it places a confirmation message on the output queue, *Qout*. Right-click the **Qout** queue, and select **Browse Messages...**.
9. Select the last displayed message within the list, and click **Properties**. This displays the properties for the message.
10. Select the **Data** tab to view the content for the message, and verify that the stock level was increased for the item (see Figure 4-21).

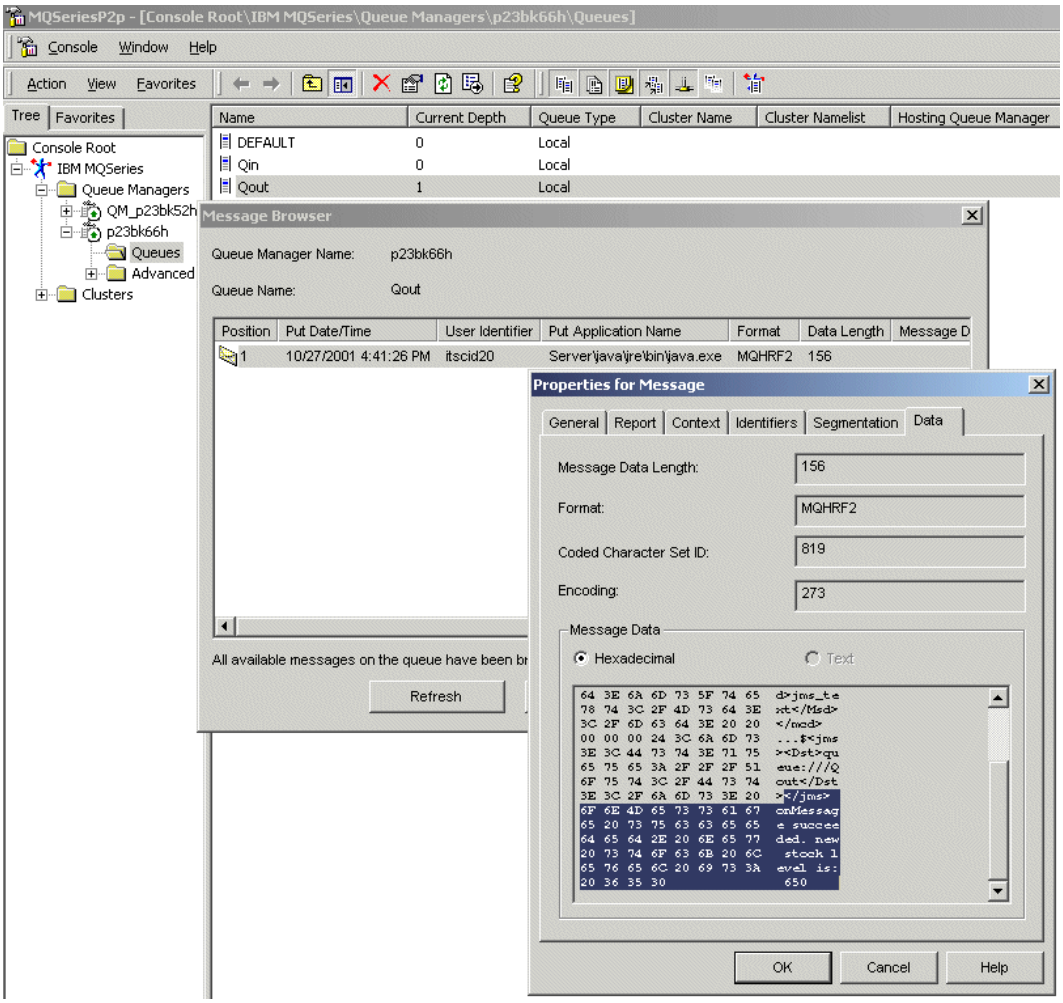


Figure 4-21 Confirming the test message was successfully handled

4.3 JMS publish/subscribe

Publish/subscribe can be considered an extension of the point-to-point messaging model. JMS destinations are no longer *queues*, but *topics*. Any application that needs to get a message of the topic has to register with it. In addition to the components in the point-to-point model, the publish/subscribe model introduces MQSeries message broker.

A Java application creates a connection to a topic and publishes a message there. The broker filters the message to all listeners registered with that topic. From here, it is similar to the point-to-point model; JMS Listener then invokes message beans in accordance with its XML configuration file (see Figure 4-22).

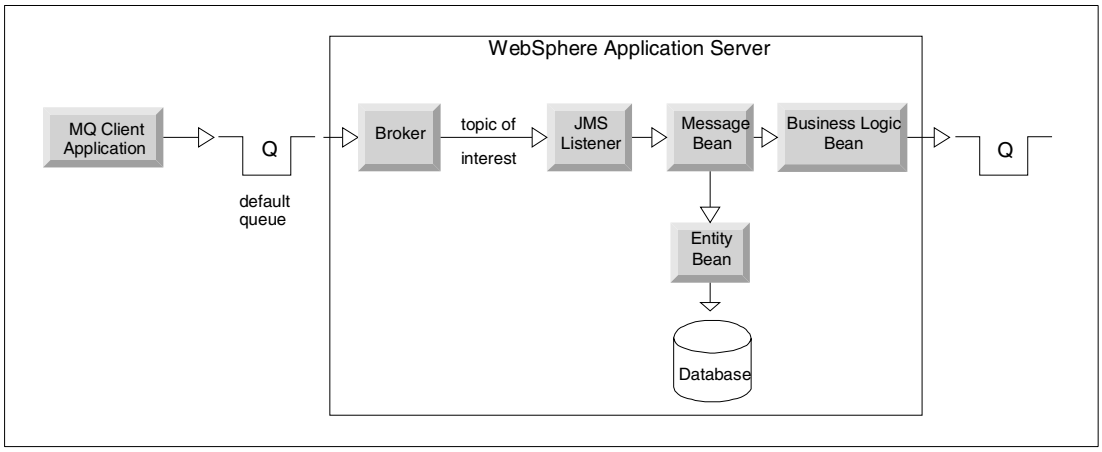


Figure 4-22 The JMS publish/subscribe model

The architecture for publish/subscribe differs from the point-to-point model in several key aspects. These are discussed in the following section.

4.3.1 The core components of JMS publish/subscribe

There are five core components for the JMS publish/subscribe model as shown in Figure 4-23.

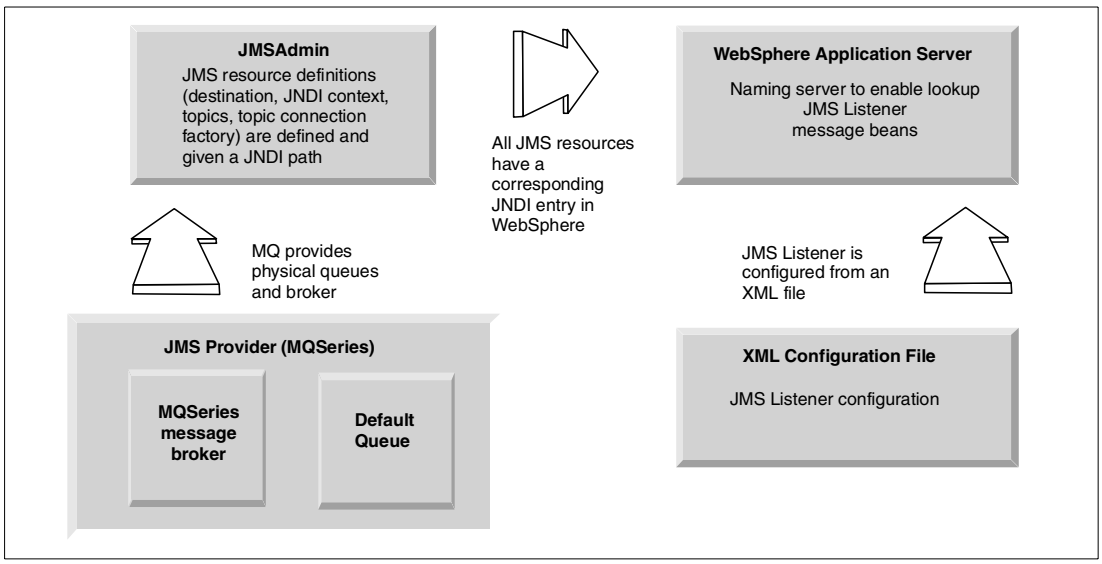

JMSAdmin
JMS resource definitions (destination, JNDI context, topics, topic connection factory) are defined and given a JNDI path
All JMS resources have a corresponding JNDI entry in WebSphere
WebSphere Application Server
Naming server to enable lookup JMS Listener message beans
MQ provides physical queues and broker
JMS Listener is configured from an XML file
JMS Provider (MQSeries)
MQSeries message broker
Default Queue
XML Configuration File
JMS Listener configuration

Figure 4-23 The core components of JMS in the publish/subscribe model

The JMSAdmin tool is used to declare topics, rather than queues. The MQSeries message broker is used to enable publish/subscribe. WebSphere Application Server is now used as a JNDI server for topic JMS resources. The XML configuration file links message beans to the corresponding JMS Listeners. These components are discussed in the following section.

4.3.2 MQSeries message broker

To alleviate the complexity of a multiple queue manager topology, MQSeries introduces the concept of *message brokers* with the MQSeries Integrator product.

The message broker, responsible for distributing incoming messages from a JMS destination, must be started on a command line as shown in Figure 4-24.

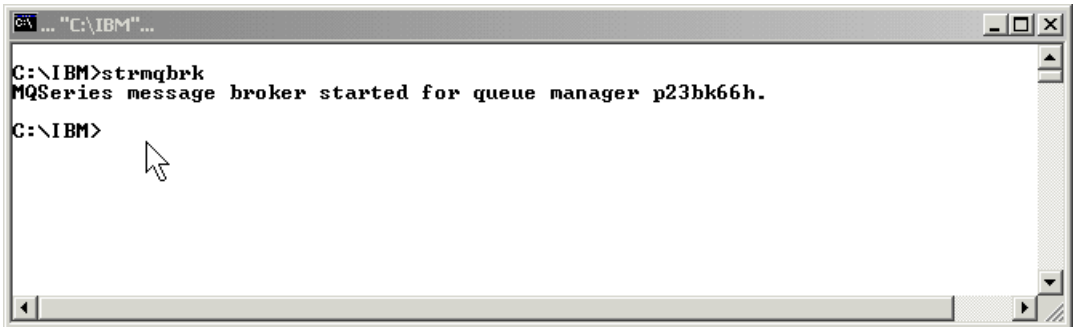


Figure 4-24 Starting MQSeries message broker

It is installed from support pack MA0C.

Note: The installation instructions for the support pack are contained in *MQSeries Using Java,* SC34-5456. They are also available from the MQSeries Web site:
http://www.ibm.com/software/ts/mqseries/txppacs/ma0c.html

4.3.3 The JMSAdmin tool

The JMSAdmin tool relationship to WebSphere Application Server and MQSeries is shown in Figure 4-25. It performs the same role as with the point-to-point model. JMS destination queues are replaced by JMS destination topics.

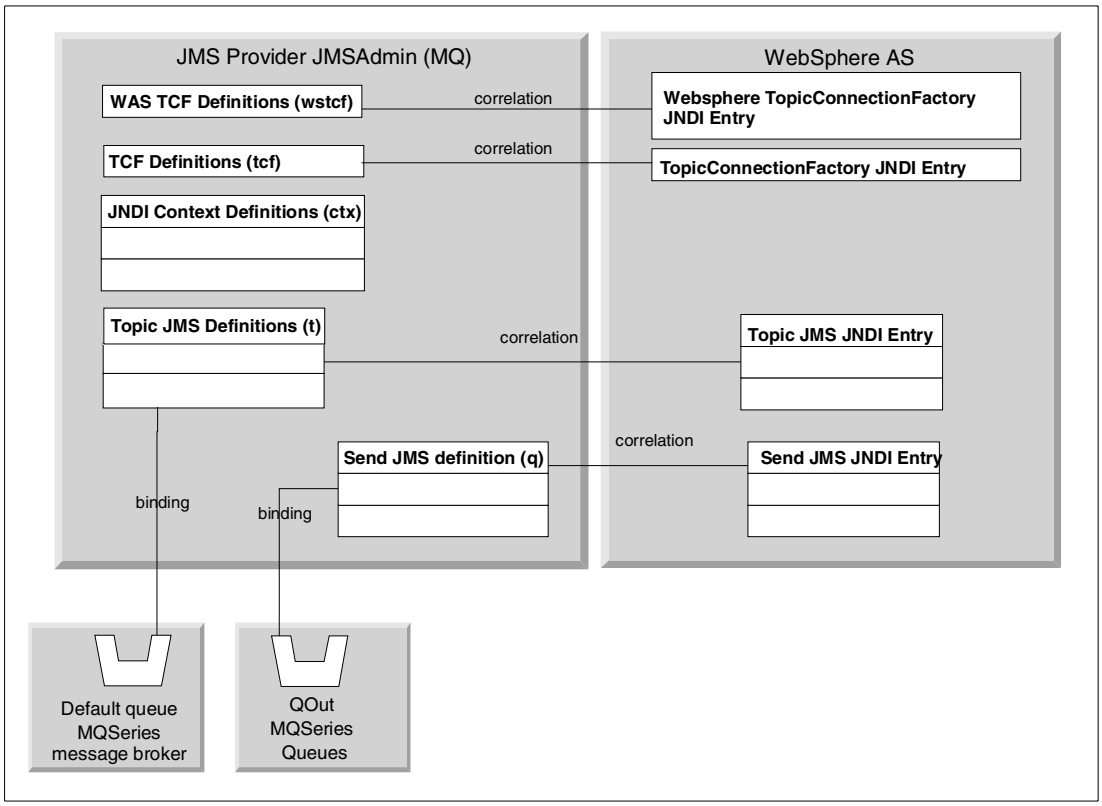


Figure 4-25 The JMSAdmin tool in the publish/subscribe architecture

An input file, such as the one shown in Example 4-7, can be used to specify all JMS resources in the JMSAdmin tool. This configuration defines three topics: *news*, *sport*, and *weather*. A listener topic, *listen*, is defined. All messages arriving for any topic in the OrderEntry JNDI path (OrderEntry/*) are routed to this JMS Listener.

Example 4-7 JMS resources defined using a JMSAdmin script, JMSpsSetup.scp

```
#
# Example script to add administered objects for the JMS pub / sub into a
# JNDI namespace.
# Apply this script using the command
# JMSAdmin < JMSpsSetup.scp
#
# Create the jms subcontext. \OrderEntry\JMS\
#  If the subcontext already exists the def command will generate an
#  "unable to create context" error which may be ignored.
#
Def ctx(OrderEntry)
#
chg ctx(OrderEntry)
#
Def ctx(JMS)
#
chg ctx(JMS)
#
# Define the TopicConnectionFactory for the client
# This uses all the default settings which map to the default queue manager.
# A non-default queue manager can be specified using the qmgr(name) property.
#
Def TCF(TCF)
#
# now define a TopicConnectionFactory for the Listener. Note the use of a WebSphere
# TCF
#
Def wstcf(TCFBean)
#
# Define the Topics, naming the Topics to be used
#
Def t(news)  topic(OrderEntry/news)
Def t(sport)  topic(OrderEntry/sport)
Def t(weather) topic(OrderEntry/weather)
#
# and the general topic used by the listener
#
Def t(listen) topic(OrderEntry/*)
#
#
# Display the current context
#
dis ctx
#
end
```

The JMS destinations

In the publish/subscribe model, Java applications publish messages to topics. The MQSeries message broker routes these to interested subscribers. JMS Listener is a subscriber (refer to Example 4-7). Three topics – news, sports, and weather – are defined. The *listen* topic, given a JNDI path of OrderEntry/*, acts as a wild card. Posting a message to news, sports, or weather results in the JMS Listener, *listen*, responding to that message.

Topic connection factory

The topic connection factory is an interface used by a Java application to create topic connection objects. These objects allow messages to be placed on that topic. It is used by the JMS Listener to retrieve messages from the topic. The messages are then passed to message beans associated with that JMS Listener.

Two types of topic connection factories are available:

- **A standard, non-transactional connection factory**: Defined in the JMSAdmin tool as TCF.
- **A transactional connection factory**: Defined as WSTCP in the JMSAdmin tool that supports transactional messaging, where messages can be placed back on the topic in the event of message bean processing failure.

4.3.4 WebSphere Application Server

The components found on WebSphere Application Server are:

- **JMS Provider**: This requires configuration identical to that for point-to-point.
- **JNDI entries**: Publish/subscribe uses topic connection factories and topics in place of queues as JMS destinations. Messages are published to the JMS destination. As in the case of point-to-point messaging, all JMS resources have a corresponding JNDI entry in the WebSphere Application Server. WebSphere Application Server is used for all JNDI lookup.
- **JMS Listener**: For the publish/subscribe messaging model, the configuration for the JMS Listener within WebSphere Application Server must have an XML configuration file containing publish/subscribe entries as shown in Figure 4-26. The JMS Listener support is enabled in the same way as for the point-to-point model (see “JMS Listener” on page 63).

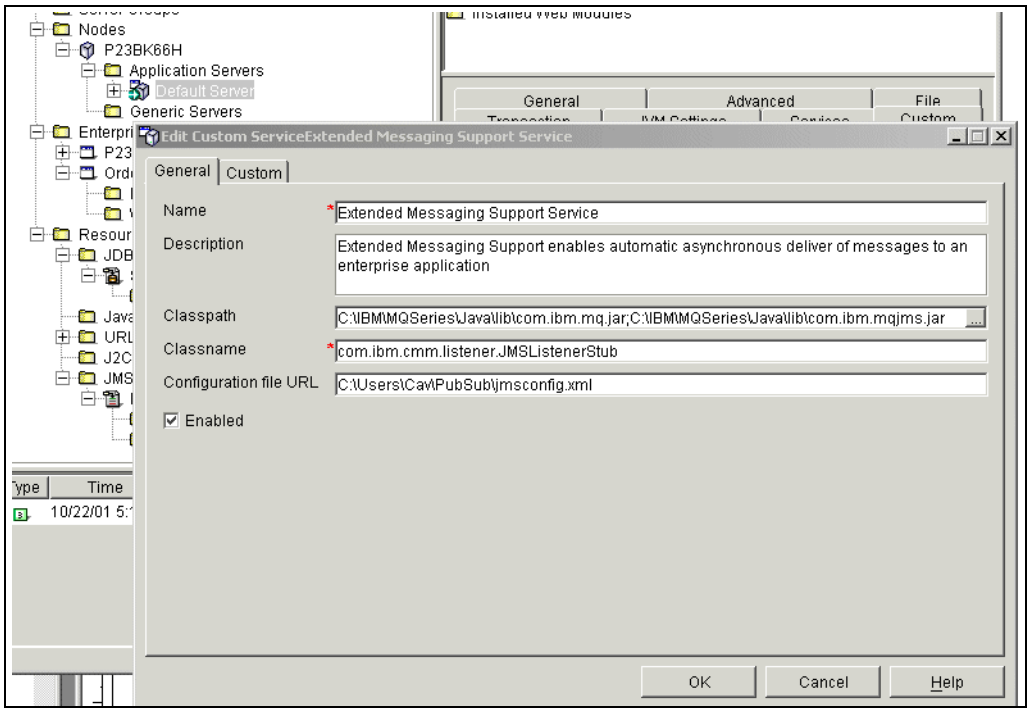


Figure 4-26 Configuring the JMS Listener service for the publish/subscribe model

4.3.5 JMS Listener XML configuration file

The XML configuration file determines which message beans are invoked for a message arriving on a given topic. A typical configuration file is shown in Example 4-8, where:

- **JMSDestination** specifies the topic, or topics if using a wildcard, to which the JMS Listener responds.
- **JMSDestinationType** specifies the type of JMS Destination that this listener stanza is for. In our example, it is topic.
- **JMS ConnectionFactory** specifies the connection factory, via its JNDI name, used for this JMS Listener.

Example 4-8 A typical JMS Listener XML configuration file for the publish/subscribe model

```
<Config>
<Pooling>
 <Timeout>100636</Timeout>
 <Threshold>36</Threshold>
</Pooling>
<Listener>
 <HomeJNDIName>OrderEntry/JMS/StockDeliveryUpdate</HomeJNDIName>
 <JMSConnectionFactory> OrderEntry/JMS/TCFBean </JMSConnectionFactory>
 <JMSDestination> OrderEntry/JMS/listen </JMSDestination>
 <JMSDestinationType> javax.jms.Topic </JMSDestinationType>
 <JMSSubscriptionDurability> durable </JMSSubscriptionDurability>
 <JMSSubscriptionName> Listener </JMSSubscriptionName>
 <MaxRetries>6</MaxRetries>
 <MaxSessions>1</MaxSessions>
</Listener>
</Config>
```

4.3.6 The message bean for the publish/subscribe model

The same rules for the message bean in the point-to-point model apply to the publish/subscribe model. Refer to 4.2.6, “The message bean” on page 67.

Placing a message at a topic JMS destination

Example 4-9 shows a *publishMessage()* method. This method can be used for publishing messages to a JMS destination topic. The objects for publishing to a topic differ from those used to send messages to a queue in these key areas:

- A topic connection factory is used to create a `TopicConnection` object.
- The topic connection object is used to connect to a JMS destination topic.
- The session is maintained through a `TopicSession`.
- A topic publisher is used to publish to the *`Topic`* topic.

The difference in the variable types used to *publish*, rather than *send* a message can be seen. Compare it to Example 4-6 on page 72.

Example 4-9 Placing a message at a JMS destination topic – The publishMessage() method

```
   private void publishMessage(String TargetTopicJNDIName, String msgID, String text) {
      TopicConnectionFactory oTopicConnectionFactory     = null;
      TopicConnection        oTopicConnection      = null;
      TopicSession           oTopicSession = null;
      TopicPublisher    oTopicPublisher = null;
      Topic           oTopic = null;

      try {
         Context oInitialContext = getInitialContext();

          //
          // Attempt to retrieve a TopicConnectionFactory from the JNDI namespace
          //
          oTopicConnectionFactory = (TopicConnectionFactory)oInitialContext.lookup(
"OrderEntry/JMS/TCF" );
          if (oTopicConnectionFactory == null)
          {
             System.out.println( "Failed to retrieve a TopicConnectionFactory from JNDI");
          }

          //
          // Get the Topic (where to publish our message) from JNDI
          //
          System.out.println("Retrieving Topic from JNDI");

          //
          // Attempt to retrieve Topic from the JNDI namespace
          //
          oTopic = (Topic)oInitialContext.lookup( TargetTopicJNDIName );

          //
          // Okay, we have completed the resource lookup, now send the message.
          // ------------------------------------------------------------------
          //

          //
          // Create a TopicConnection from the TopicConnectionFactory
          //
```

```
System.out.println("Creating a Connection");
oTopicConnection = oTopicConnectionFactory.createTopicConnection();

System.out.println("Created a Connection");


//
// IMPORTANT: Receive calls will be blocked if the connection is
// not explicitly started, so make sure its started.
//
System.out.println("Starting the Connection");
oTopicConnection.start();

//
// Create a TopicSession from the TopicConnection.
// Not transacted, and that it should
// automatically acknowledge received messages
//
System.out.println("Creating a Session");

boolean transacted = false;
oTopicSession = oTopicConnection.createTopicSession( transacted,
                Session.AUTO_ACKNOWLEDGE);
//
// Use the session to create a TopicPublisher, passing in the
// destination (the Topic object) as a parameter
//
System.out.println("Creating a TopicPublisher");
oTopicPublisher = oTopicSession.createPublisher(oTopic);

//
// Use the TopicSession to create messages, create an empty TextMessage
// and add the data passed.
//
System.out.println( "Creating a TextMessage" );
TextMessage outMessage = oTopicSession.createTextMessage(text);

//
// Ask the TopicPublisher to publish the message we have created
//
System.out.println( "Publish the message to " + oTopic.getTopicName() );
oTopicPublisher.publish(outMessage);

//
// Closing TopicPublisher
//
System.out.println("Closing TopicPublisher");
oTopicPublisher.close();

//
// Closing Session.
//
System.out.println("Closing Session");
oTopicSession.close();
oTopicSession = null;

//
// Closing Connection.
//
System.out.println("Closing Connection");
```

```
            oTopicConnection.close();
            oTopicConnection = null;

            System.out.println("End of Sample");

        }
        catch (JMSException je)
        {
            System.out.println("JMS failed with "+je);
            Exception le = je.getLinkedException();
            if (le != null)
            {
                System.out.println("linked exception "+le);
            }
        }
        catch( Exception exc )
        {
            System.out.println("Unexpected error " + exc);
        }
        finally
        {
            //
            //  Ensure JMS objects are closed
            //
            try
            {
                if (oTopicSession != null) {
                    System.out.println("Closing session");
                    oTopicSession.close();
                }
               if (oTopicConnection != null) {
                     System.out.println("Closing connection");
                   oTopicConnection.close();
               }
            }
            catch (JMSException je)
            {
                je.printStackTrace();

                System.out.println("Unexpected error, failed with "+je);
            }
        }
    }
```

4.4 Advanced JMS facilities

This section discusses advanced JMS facilities that need to be configured for high performance, secure and robust messaging infrastructure.

4.4.1 Transaction support

The transactional behavior of the message read from the JMS destination can be configured for each listener stanza in the XML configuration file.

If transactional handling is specified, before the message is read from the destination, a global transaction is started by JMS Listener. The transaction is committed or rolled back when the *onMessage()* method has completed.

The removal of the message from the destination is directly associated with the completion of the transaction. If the transaction is marked for rollback only, the rollback of the transaction will cause the message to remain on the destination. The commit of the transaction will cause the message to be removed from the destination.

Use the application assembly tool or WebSphere Studio Application Developer to configure the *onMessage()* method of the message bean to support the transactional behavior that you want. For future compatibility with the EJB2.0 specification, message beans specify either bean-managed transactions (BMT) or container-managed transactions (CMT) with a transaction behavior of TX_NOT_SUPPORTED or TX_REQUIRED.

When transaction handling is not specified, the message is read from the destination queue without being part of a global transaction.

For transactional behavior, the WebSphere-specific JMS/XA connection factories – *wsqcf* for queue and *wstcf* for topic – must be defined in the JNDI name space using the JMSAdmin tool. For destinations not requiring global transaction support, the JNDI name space can be configured using the non-XA JMS connection factories such as *qcf* for queue and *tcf* for topic. With non-WebSphere XA connection factories, JMS calls issued on the destination are not part of WebSphere Application Server transactions. Also JMS resources are not recovered if the server fails.

Bean-managed transactions (BMT)

The configuration file should not specify transactions for a message bean that uses bean-managed transactions. It is not really a restriction, but more of a recommendation. The system starts a transaction to do the initial receive, but the message bean is not part of the same transaction if it is using bean-managed transactions.

Container-managed transactions (CMT)

Only two of the six transactional policies are meaningful for a message bean since a client is not involved and the *onMessage()* method is the originator of the work request to the server. The two policies are:

- **Required**: The configuration file for the message bean should also specify `Transactional=True`. This ensures the message is read within a transaction and the processing within the *onMessage()* method occurs within the same transaction (refer to Table 4-1).
- **Not Supported**

Table 4-1 EJB transaction attributes

CMT behavior	Transactional=True	Transactional=False
EJBs with CMT and *Required*	The initial read and execution of *onMessage()* are both part of the same transaction.	Global transaction is not started
EJBs with CMT and *Not Supported*	The initial read is within a transaction, but the *onMessage()* method is not part of the same transaction.	Global transaction is not started
EJBs with Bean Managed Persistence (BMT)	The initial read is within a transaction, but the *onMessage()* method is not part of the same transaction.	Global transaction is not started

If the Transactional value is set to “False”, the message is not read as part of a transaction, although the *onMessage()* method may be transactional depending upon the method properties.

4.4.2 JMS Listener threads and MaxSessions

MaxSessions is a listener attribute within the listener stanza of the XML configuration file. This setting has a direct impact on overall system performance.

Each application server that has Extended Messaging configured and at least one JMS Listener defined in its XML configuration file has a pool of threads defined internally. These threads take messages off of the queue, obtain the associated message bean instance, and call the *onMessage()* method for that bean. This pool of threads is referred as the *Server Session Thread Pool*.

The Server Session Thread Pool begins with five threads and has a maximum sustained thread pool size of 20. Once the maximum pool size is reached, any additional threads used are not pooled; they are disposed. It is likely that the next release of WebSphere will allow configuration of this pool. Only *one* Server Session Thread Pool exists for a WebSphere Application Server, regardless of how many listener stanzas exist for a given server.

In the simple case, where each WebSphere Application Server has one listener stanza, the current values for MaxSessions is sufficient. In a more complex case where the number of listener stanzas is greater than one, each listener competes for the same threads from the Session Thread Pool. For example, if you defined 10 JMS Listeners, on average, there are only two Server Session Thread Pool threads for each JMS Listener. In some cases, it will be necessary to configure multiple application servers, each with a set of listeners on a given machine or node. This relaxes the constraint of 20 threads in a Server Session Thread Pool, and in fact, multiplies it by the number of servers defined for a given node.

Continuing to add servers until maximum resource utilization of the machine is achieved ensures maximum efficiency of the JMS Listener service. It is possible that, at some point, adding additional servers no longer increases throughput.

4.4.3 Exception handling

Exception handling for a message bean is unique due to asynchronous processing of the message. A failure cannot be reported to a client program. Exceptions should not be returned from the *onMessage()* method, but if this occurs, they must be handled by the container.

Any exception returned from the *onMessage()* method causes the message to remain on the JMS destination. JMS Listener retries. If the situation that caused the exception still exists, the message remains on the destination. Therefore, JMS Listener will be able to read the message again from the destination.

This action may not be desirable. The maximum number of times that JMS Listener reads the message from the queue is configured via the MaxRetries attribute of the XML configuration file. When the maximum number of reads has occurred, the JMS Listener for that specific destination is stopped. Once the problem is resolved, it is necessary for the associated EJB jar to be stopped and restarted using the WebSphere Application Server Administrative Console to continue handling messages from the destination.

When the message that removed from the destination is part of a global transaction, any exception in the *onMessage()* method has the effect of marking the transaction as *rollback only*. The transaction rolls back, causing any work performed on the global transaction to rollback, which includes leaving the message on the queue.

The basic structure of the *onMessage()* method should follow one of the following forms:

- Message bean is part of a global transaction. The exception is caught in the *onMessage()* method. The transaction is marked as rollback only. This causes the transaction to be rolled back at commit time. The message is placed back on the queue and processed again until either the transaction successfully commits or the maximum number of retries is reached. At this time, the listener is stopped. No additional messages are processed until the problem is fixed.

```
public void onMessage(javax.jms.Message msg) {
   try {
      ... // call to business logic
   } catch (java.lang.Throwable exception) {
      // Mark the transaction as rollback only
      tx.setRollbackOnly();
   }
}
```

- Message bean is not part of a global transaction. Since there is no transaction active, the application is responsible for error handling in the *onMessage()* method. Returning the exception to the container would cause the message to be reprocessed. If the problem would persist, the exception would again be returned. The message could be placed on another queue of unprocessed messages. The message could later be inspected by a system administrator, or processed by another message bean. Since the *onMessage()* is not part of a transaction, the results of partially processed messages may persist depending upon the application.

```
public void onMessage(javax.jms.Message msg) {
   try {
      ... // call to business logic
   } catch (java.lang.Throwable exception) {
      // put message on unprocessed queue
      unprocessedQueue.put(msg);
   }
}
```

  Putting the message to another queue can also be used for a message that is part of a global transaction. This prevents the listener from stopping and permits it to continue processing messages. This is especially desirable behavior for handling application exceptions when messages, for example, are not formatted as expected.

  There is an alternative implementation for exception handling. It is similar to the previous example. That is, the transaction configuration is the same, and the message bean is not part of a global transaction.

```
public void onMessage(javax.jms.Message msg) {
   try {
      ... // call to business logic
   } catch (java.lang.Throwable exception) {
      // Potentially, the message bean may need
      // compensation logic to undo things that
      // were done by downstream transaction
      doRecovery();

      // return the exception to the container
      throw exception;
   }
}
```

  A combination of these techniques may be implemented.

- The message itself contains the number of retry attempts that have been made:

  ```
  int retry = msg.getIntProperty("JMSXDeliveryCount");
  ```

  The returned value can be used to determine the exception processing that should take place.

4.4.4 Security

Messages arriving at a destination being processed by the JMS Listener have no client credentials associated with them. The messages are anonymous. In the current release, the security model provides a solution where a JMS Listener assumes the EJB server process credentials for the invocation of the stateless session bean.

In configuring the secure WebSphere Application Server, it is necessary to configure:

- Resource security for the stateless session bean to allocate its security method groups
- Security permissions for the bean *create()* and *onMessage()* methods so they are accessible by the given EJB server principal

This security model is to be reviewed for the next release.

4.4.5 MQSeries clustering

Message distribution, managed through the use of MQSeries clusters, is an efficient way to manage point of failures in the message network. The cluster manager can detect failure with the physical machine, or the message queue availability, and route messages to queues that are available. It can also detect when a queue is full and no longer accepts messages. If all physical queues are full on the system, the control point continues to accept messages. The messages are stored on an internal queue until the physical queues can accept messages.

In an advanced configuration, the MQSeries cluster control point can contain a modified exit routine that can be called to control the distribution of messages to the physical queues.

Another configuration value that can be used in the control of messages is to intentionally set the physical size of the queue depth to a relatively small number. The cluster control point would control messages to the queues, if there was a problem with MQSeries, WebSphere Application Server, or a system failure, the control point would manage the returning of messages to the available systems. This technique would allow for an earlier detection of a WebSphere Application Server failure. It would prevent a large number of messages from being orphaned on a machine when a failure occurs. It would also allow for messages that occurred close to the failure to be handled by an alternate system.

Publish/subscribe model clustering restriction

The use of MQSeries publish/subscribe in an MQSeries cluster with cluster queues is not a practical configuration. This is because a subscriber has an identity, which in MQSeries is identified by the combination:

```
subscriber queue name + Queue manager name + Correlation ID
```

Since a JMS subscriber only receives publications that have its correlation identifier, and these are unique, a given publication message must be delivered to its defined subscriber. This means subscription queues should not be cluster queues when the hosting queue manager is part of a cluster.

4.5 The future of WebSphere messaging

WebSphere Application Server Enterprise Edition Version 4.0 combines the basic JMS and JMS/XA support of WebSphere Advanced Edition 4.0 with support for JMS Listeners and message beans. This is sufficient to begin deploying applications that integrate legacy systems via messaging and new systems that are coupled by asynchronous means.

The next release of WebSphere Application Server Enterprise Edition will implement J2EE 1.3 and EJB 2.0 specifications. This will include a certified support for message-driven beans. This message-driven bean implementation will be an evolution of what is available in WebSphere Application Server Enterprise Edition now. Modifications to the EJB container, to ensure compliance with EJB 2.0 message-driven beans, is underway now. Tooling for creation and deployment is being modified to leverage the Message-Driven Bean deployment descriptor format.

5

The WorkArea service

WorkArea provides a storage area where multiple pieces of data can be stored, and all the data is passed along transparently by the WebSphere Application Server. This storage is associated with a single thread. Data is stored as a name-value-mode triplet. This chapter describes:

- When to use WorkArea
- How the WorkArea service works
- Using WorkArea in the example application
- Ways to use WorkArea beyond the example application
- How to configure the WorkArea service in the WebSphere Application Server

Note: To understand the structure of the OrderEntry application and to run it, read the instructions in Appendix A, “Downloading and installing the OrderEntry application” on page 221.

© Copyright IBM Corp. 2002

5.1 WorkArea overview

WorkArea is best used to store data that is required by many parts of the application. If we pass this data as parameters, method signatures would quickly become cluttered. Even methods that don't need the data may have to pass it along to other methods that require it. Also, if future enhancements require additional data to be passed or the type or length of the data must change, all methods that use or have to pass this data would have to change.

By using WorkArea, only those methods that actually use the data would have to handle it. Because WorkArea is not explicitly passed, there are no extra parameters for methods that don't need the data. In this sense, WorkArea is similar to the security or transaction context in WebSphere Application Server.

Each piece of data can be accessed separately, so if the type or length of the data changes, only those methods that actually use that particular piece of data are affected by the change. In addition, the ability to modify the data in WorkArea can be controlled by setting modes for each piece of data.

The information in WorkArea consists of a set of properties; a property consists of a *key-value-mode triplet*. The key-value pair represents the information contained in the property; the key is a name by which the associated value is retrieved. The mode determines whether the property can be removed or modified.

The applications access the WorkArea facility by using the `com.ibm.websphere.workarea.UserWorkArea` interface. It contains nine methods that we discuss throughout this chapter.

WorkArea has an important characteristic. The data can only be passed one way, from the caller to the remote method. If the remote method makes changes to the data, the changes are never seen by the caller. A remote method can be another servlet or a method in the remote interface of an EJB. Even if the servlet or bean resides in the same JVM, the call is still considered remote. It is important to remember this characteristic when designing applications that will use WorkArea.

For the remainder of this chapter, we refer to a piece of data in WorkArea as a *property*.

5.1.1 WorkArea scope

An important concept to understand about WorkArea is the scope. WorkArea is associated with a thread. Information is passed in WorkArea from the caller to a remote method, which is done using one-way communication. The remote method is not allowed to change, add, or remove properties in WorkArea created by the caller. Attempting to use the set method on WorkArea that was not created by the current component results in a `com.ibm.websphere.workarea.NotOriginator` exception.

To add a new property or override the properties in an existing WorkArea that was initiated in another method, the remote method must create a new WorkArea that is associated with the existing one. A new WorkArea is called *nested WorkArea*. Creating and using a nested WorkArea is described in more detail in 5.3.2, "Nested WorkAreas" on page 97.

WorkArea may be created by a client and then passed transparently to a server, or it may be created by the server and passed transparently between servlets and EJBs on the same or different servers. In any case, WorkArea behaves in the same way. We refer to method calls as remote, even though for the example application, the calls are actually made on the same server. The only time WorkArea can be considered to be passed locally is when the servlet or bean makes a call to a local helper class that is not a servlet or EJB.

WorkArea is associated with a single thread of execution. WorkArea is passed from the client thread to the server. However, WorkArea is not transferred to new threads spawned from the thread that WorkArea is associated with. For this reason, WorkArea must be used cautiously in applications that use the Java Abstract Windowing Toolkit (AWT). The AWT implementation is multithreaded, and WorkArea begun on one thread is not available on another. For example, if a program begins WorkArea in response to an AWT event, such as pressing a button, WorkArea may not be available to any other part of the application after the execution of the event completes.

5.2 Using WorkArea in the example application

In the example application, we use WorkArea to pass information between the different parts of the application without having to explicitly pass a parameter. We could just use the `request` object to pass the information, but we would have to explicitly pass the `request` object when it is used outside the *service* method of the servlet. This can be cumbersome since there is much more information in the request than we need. It can also lead to errors that are hard to find since there is no mechanism to prevent changing the properties in the request object.

To use WorkArea, we must perform the following actions in the application:

1. Bind to the WorkArea facility. This is done using JNDI. Once we bind a `com.ibm.websphere.workarea.UserWorkArea` object to the facility, this same object can be used to access many WorkAreas. The actual WorkArea that the methods operate on is the one associated with the current thread.
2. Call the *begin* method. This method creates WorkArea. If WorkArea is already associated with the current thread, a nested WorkArea is created inside of it. This step is not necessary if we only want to read the properties in an existing WorkArea and not change them.
3. Use the *set* and *get* methods to access and change properties in WorkArea.
4. Call the *complete* method. This method terminates the current WorkArea. This must only be done in the same servlet or bean that called the *begin* method.

5.2.1 Binding to the WorkArea facility

First, we bind to the WorkArea facility using JNDI. To do this, we need to access an initial context object. Example 5-1 shows binding to the facility in the *init* method of `OrderEntryServlet`.

Example 5-1 Binding to the WorkArea facility

```
import com.ibm.websphere.workarea.*;
import javax.naming.*;
...

private UserWorkArea workArea;
...

public void init(ServletConfig config) throws ServletException
{
   super.init(config, SuperServlet.SYSTEM);
   InitialContext jndi = null;

   // Bind to WorkArea facility
   try {
```

```
        jndi = new InitialContext();
        workArea = (UserWorkArea)jndi.lookup("java:comp/websphere/UserWorkArea");
    }
    // This exception indicates a problem binding to the service
    catch (NamingException e) { e.printStackTrace(); }

    flexLog("OrderEntryServlet: init");
}
```

Once we bind to the facility, calls to the *WorkArea* methods locate the WorkArea associated with the current thread. Notice that we do not call the *begin* method here; that is done from the *doPost* method as described in the next section. The lookup binds to the facility that maintains WorkAreas. The *begin* method creates a new or nested WorkArea associated with the current thread. A reference to the WorkArea facility can work for different threads, but calls to the *service* method of a servlet and calls to an EJB can come from different threads. We must only call the *begin* and *complete* methods within the scope of a single thread.

5.2.2 Creating and populating WorkArea

We create and populate WorkArea in the *doPost()* method of the `OrderEntryServlet` by invoking the *createWorkArea()* method. All the information needed in other parts of the application is stored in WorkArea. Invoke the *begin* method to create a new WorkArea and associate it with the current thread. The *begin* method takes a `String` argument that is used as the name of WorkArea. This name is used primarily for debugging.

In this case, we used the servlet name to identify the servlet that created WorkArea. The only requirement for the name of WorkArea is that it is not null. To set the properties, we use the *set* method and specify the name and value of the property. Example 5-2 shows the method that creates WorkArea and sets the values.

Example 5-2 Creating and populating WorkArea

```
import com.ibm.websphere.workarea.*;
...

private UserWorkArea workArea;
...

private void createWorkArea( HttpServletRequest request ) {

    // Create a new WorkArea associated with this thread
    workArea.begin("OrderEntryServlet");

    try {

        // Get the customer information from the request
        String id = request.getParameter("customerID");
        String lastName = request.getParameter("lastName");
        String firstName = request.getParameter("firstName");

        // Store the customer information in WorkArea
        workArea.set("customerID", id);
        workArea.set("lastName", lastName);
        workArea.set("firstName", firstName);

        // Get the item information from the request
        String itemID = request.getParameter("itemID");
```

```
      String quantity = request.getParameter("quantity");

      // Store the item information in WorkArea
      workArea.set("itemID", itemID);
      workArea.set("quantity", quantity);
   }
   // Since we did not specify read only properties, this exception will not occur here.
   // If this exception were caught, it would indicate a coding error.
   catch( PropertyReadOnly e ){ e.printStackTrace(); }

   // WorkArea was created inside this method, neither of these exceptions will occur.
   // NoWorkArea would indicate WorkArea was not created.
   catch( NoWorkArea e ){ e.printStackTrace(); }
   // NotOriginator would indicate WorkArea was created by some other servlet or bean.
   catch( NotOriginator e ){ e.printStackTrace(); }
}
```

Now the information from the request is available to the rest of the application without having to explicitly include the `request` as a parameter.

5.2.3 Accessing WorkArea

To access WorkArea in other parts of the application, we add a method to the `ApplicationMgr` and `OrderEntryClerkBean` classes to bind to the WorkArea facility. This method uses the same `Context` that is used to bind to the `OrderEntryClerk` EJB. Example 5-3 shows the method that binds to the WorkArea facility.

Example 5-3 Accessing WorkArea

```
import com.ibm.websphere.workarea.*;
import javax.naming.*;
...

private UserWorkArea getWorkArea(Context ctx){

   UserWorkArea workArea = null;
   try {
      workArea = (UserWorkArea)ctx.lookup("java:comp/websphere/UserWorkArea");
   }
   // This exception indicates a problem binding to the facility
   catch( NamingException e ) { e.printStackTrace(); }

   return workArea;
}
```

Once we bind to the WorkArea facility, the `UserWorkArea` object is used to access the WorkArea associated with the current thread. There is no need to specify which WorkArea we want. There can be only one WorkArea associated with a thread. There is no way to access WorkArea associated with another thread.

5.2.4 Using WorkArea

To get information out of WorkArea, use the *get* method with the name of the property you want to get. Example 5-4 shows how to retrieve the customer information from WorkArea.

Example 5-4 Retrieving values from WorkArea

```
import com.ibm.websphere.workarea.*;
...

Context ctx = getInitialContext();
UserWorkArea workArea = getWorkArea(ctx);

String id = (String)workArea.get("customerID");
String lastName = (String)workArea.get("lastName");
String firstName = (String)workArea.get("firstName");
```

Notice that the *get* method does not throw any exceptions. If no WorkArea was associated with the current thread, or it did not contain a property with the name specified, the *get* method returns null.

5.2.5 Terminating WorkArea

Always terminate WorkArea when finished using it. This must be done by the same process that created WorkArea. Attempting to call the *complete* method outside the bean or servlet that called the *begin* method will result in a `NotOriginator` exception. Since this exception indicates an application coding error, we include the WorkArea name in the information written to the log. We get the name by calling the *getName* method. If we always use the name of the servlet or bean that created WorkArea as its name, this error will be easier to debug. Example 5-5 shows a method to terminate WorkArea.

Example 5-5 Terminating WorkArea

```
import com.ibm.websphere.workarea.*;
...

private void terminateWorkArea() {

   // Complete WorkArea associated with this thread
   try {
      workArea.complete();
   }
   // Ignore this exception since we were trying to terminate anyway
   catch (NoWorkArea e) { }
   // This exception indicates that this WorkArea was created by some other servlet
   catch (NotOriginator e) {
      flexLog( "WorkArea could not be terminated by OrderEntryServlet: " +
                  workArea.getName() );
      e.printStackTrace(); }
}
```

5.3 Beyond the example

The example application shows the simplest way to use WorkArea. This section goes into more detail about other WorkArea features. If the example has all the features you need, you can safely skip this section.

5.3.1 Using property modes to protect information

There are two *set* methods for WorkArea. The two-argument method, which was used in the example, sets a default mode of normal for the property. The three-argument method takes a mode as the third argument. This mode determines whether the property being set can be modified or removed by a remote method. As stated in the previous section, in order to change, add, or remove properties in WorkArea that was received from another servlet or bean, a nested WorkArea must be created. The properties in the original WorkArea cannot be changed.

The modes control what properties can be overridden in the nested WorkArea. *Normal mode* is the default if no mode is specified when the property is set. Normal mode means the property may be changed or removed. *Fixed mode* means the property cannot be removed from WorkArea. *Read only mode* means the value of the property cannot be changed. Table 5-1 shows the result of operations performed on the properties depending on the mode that was set in WorkArea created by the caller.

Table 5-1 Permissible operations on properties with a given mode

Mode	If using *set* operation	If using *remove* operation
Normal	Property is set with a new value	Property is removed
Read-only	PropertyReadOnly exception	Property is removed
Fixed read-only	PropertyReadOnly exception	PropertyFixed exception
Fixed	Property is set with a new value	PropertyFixed exception

In the example application, we would probably want to make the customerID field both fixed and read only. Example 5-6 shows the change to the call of the *set* method to include the third parameter.

Example 5-6 Setting the mode for a property

```
import com.ibm.websphere.workarea.*;
...

Context ctx = getInitialContext();
UserWorkArea workArea = getWorkArea(ctx);

workArea.set("customerID", id, PropertyModeType.fixed_readonly);
```

These are the possible values for the mode defined by `com.ibm.websphere.workarea.PropertyModeType`:

- `PropertyModeType.normal`
- `PropertyModeType.fixed_normal`
- `PropertyModeType.read_only`
- `PropertyModeType.fixed_readonly`

5.3.2 Nested WorkAreas

Nested WorkAreas are used by remote methods to override, add, or delete properties in a WorkArea received from the client. Figure 5-1 shows the original WorkArea created by the client with four properties. The diagrams used in this chapter differ somewhat from those in other documentation to more clearly show the process of overriding the properties.

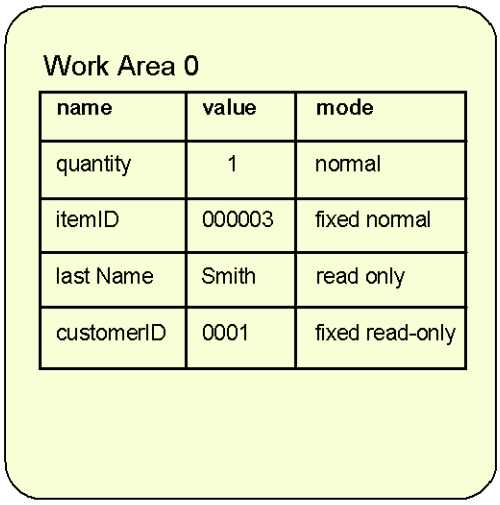


Figure 5-1 Original WorkArea

A remote method receiving this WorkArea can create a nested WorkArea that inherits all the properties of the original WorkArea received from the client. The remote method can then add, remove, and override properties in the nested WorkArea. The properties in the original WorkArea are not changed, but changes in the new WorkArea can override the properties in the original WorkArea. Property modes, discussed in the next section, control what properties can be overridden by the new WorkArea.

Figure 5-2 shows the original WorkArea 0 with nested WorkArea 1 created by a remote method. All of the properties in WorkArea 0 that are not overridden are visible in the remote method, as are all the properties in WorkArea 1. Once the `quantity` property is set with the new value 2 in WorkArea 1, the value in WorkArea 0 is no longer visible. The new value `firstName` is visible in WorkArea 1, but when the remote method returns (WorkArea 1 is terminated), it will not be accessible in WorkArea 0.

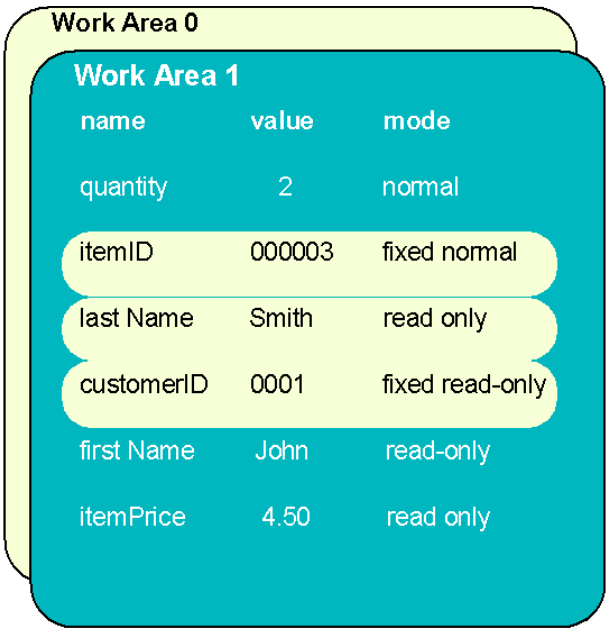


Figure 5-2 Nested WorkArea

If the remote method, in turn, calls another remote method, WorkArea 1 is passed to the next method along with the properties from WorkArea 0 that have not been overridden. That method can also create another nested WorkArea over WorkArea 1.

Figure 5-3 shows a double-nested WorkArea 2. In WorkArea 2, again, we changed the value of `quantity` to 3. This overrides the value in WorkArea 1. All the other WorkArea properties from WorkArea 1 are visible. If we remove the firstName property, it will no longer be visible in the current method. However, when control returns to the caller (method where we created WorkArea 1), the `firstName` property will again be visible.

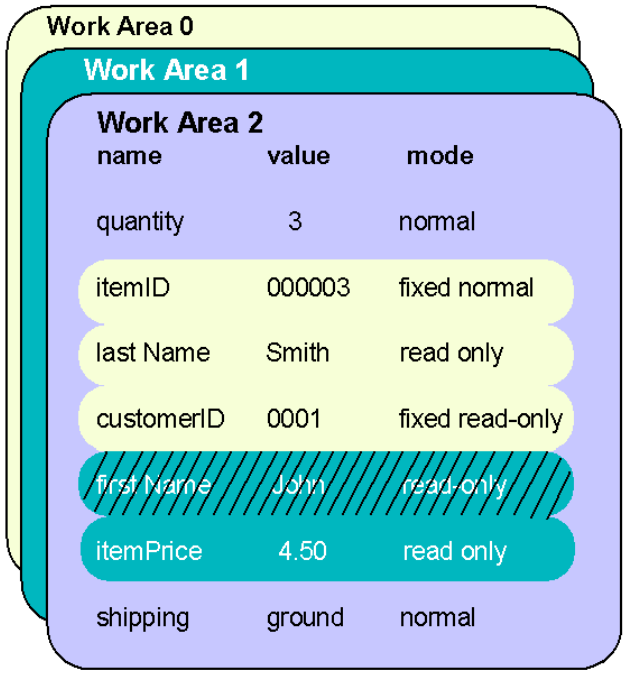


Figure 5-3 Double-nested WorkArea

Note: A very good analogy to the concept of the nested WorkAreas is the class hierarchy in Java. For example, base class `A` has three methods: `x()`, `y()`, and `z()`. We can create another class `B` that inherits from class `A`. Class `B` overrides method `z()` and defines some other methods. If we invoke method `z()` inside the object of class `B`, it finds this method implementation in class `B`. If it invokes method `x()` or `y()`, it finds their implementations in class `A`. In this example, class `A` is equivalent to the original WorkArea, and class `B` is equivalent to the nested WorkArea.

Creating a nested WorkArea

In the example, if we try to use the *set* method to override an existing property or to add a new property anywhere outside the `OrderEntryServlet`, we would get a `NotOriginator` exception. We may only use the *get* method to read values from WorkArea outside the `OrderEntryServlet`. To override a property in an existing WorkArea, that was not originated in the current servlet or bean, we must create a nested WorkArea. A nested WorkArea is created the same way as the original WorkArea was, with the *begin* method. Example 5-7 is identical to the code for creating a new WorkArea. We pass the servlet name to identify the nested WorkArea with its creator.

Example 5-7 Creating a nested WorkArea

```
import com.ibm.websphere.workarea.*;
...

Context ctx = getInitialContext();
UserWorkArea workArea = getWorkArea(ctx);

workArea.begin("AnotherServlet");
```

Using a nested WorkArea

Using the nested WorkArea is the same as using the original WorkArea. The nested WorkArea has all the properties of the original WorkArea plus any properties added in the new WorkArea. The difference is that in the new WorkArea, we can override the properties and values of the original WorkArea. To override the value of an existing property in the original WorkArea, call the *set* method passing the name of the existing property and the new value. This actually creates a new property in the new WorkArea that overrides the property in the original WorkArea. However, to the current scope, it looks as though the original property was changed. Whether the *set* succeeds depends on the property mode set in the original WorkArea.

A property can also be removed from a nested WorkArea. Whether the property can be removed depends on the property mode set in the original WorkArea. Here again, the property is not removed from the original WorkArea, but is masked by the new WorkArea so that it appears to have been removed. When control returns to the calling method that created the original WorkArea, the property is restored with its original value. Example 5-8 shows how we would remove the item quantity from WorkArea.

Example 5-8 Removing a property from WorkArea

```
import com.ibm.websphere.workarea.*;
...

Context ctx = getInitialContext();
UserWorkArea workArea = getWorkArea(ctx);

workArea.remove(“quantity”);
```

None of the changes made in a remote method that creates a nested WorkArea are propagated back to the caller. Properties removed in the nested WorkArea are still available to the calling method. All properties will have the same values that were set by the calling method.

Terminating a nested WorkArea

When finished with a nested WorkArea, call the *complete* method, just as for the original WorkArea. This terminates only the nested WorkArea. Recall that the original or nested WorkArea may only be terminated by the servlet or bean that created it.

5.3.3 Changing the modes

In a nested WorkArea, you can override the values of properties in the original WorkArea that was not marked with the read-only mode. You can also override the mode of the properties in the nested WorkArea, depending on the mode they were set with in the original WorkArea. Table 5-2 lists the mode changes that are possible in the nested WorkArea.

Table 5-2 Permissible mode changes in a nested WorkArea

Mode in an enclosing WorkArea	Possible mode change in a nested WorkArea
Normal	Normal, read-only, fixed, fixed read-only
Read-only	No changes allowed unless a property is removed first and then re-created with any of the four modes
Fixed read-only	No change allowed
Fixed	Fixed, fixed read-only

Remember that the mode is not actually changed in the original WorkArea. What happens is that the new mode in the new (nested) WorkArea overrides the mode in the original WorkArea. When the remote method completes the new (nested) WorkArea and returns, the modes return to their original values before the remote method call.

5.3.4 Other methods defined by the UserWorkArea interface

Example 5-9 shows the complete definition of the *UserWorkArea* interface.

Example 5-9 Complete definition of the UserWorkArea interface

```
package com.ibm.websphere.workarea;

public interface UserWorkArea {
   void begin( String name );
   void complete() throws NoWorkArea, NotOriginator;

   String getName();
   String[] retrieveAllKeys();
   void set( String key, java.io.Serialazable value )
      throws NoWorkArea, NotOriginator, PropertyReadOnly;
   void set( String key, java.io.Serialazable value, PropertyModeType mode )
      throws NoWorkArea, NotOriginator, PropertyReadOnly;
   java.io.Serializabel get( String key );
   PropertyModeType getMode( String key );
   void remove( String key ) throws NoWorkArea, NotOriginator, PropertyFixed;
}
```

The following methods are public methods that are available on WorkAreas:

- *retrieveAllKeys*: Returns an array of `Strings` that contain the names of all properties in WorkArea. If no WorkArea was found, the method returns null. If WorkArea contains no properties (for example, WorkArea was created but no properties were set), the method returns an array of zero length.
- *getMode*: Returns a `PropertyModeType` indicating the mode for the property with the given name.

5.3.5 Putting other objects in WorkArea

It is possible to store objects other than `Strings` in WorkArea. These objects must implement the *`java.io.Serializable`* interface. `Strings` and some other commonly used objects already implement `Serializable`. This interface requires two methods, *writeObject* and *readObject*. The *writeObject* method writes the object to an `ObjectOutputStream` and *readObject* method reads the object from an `ObjectInputStream`. This is usually implemented by writing and reading the individual fields of the object. For objects that only contain only primitives, the default implementation can be used. See the API documentation for the *`java.io.Serializable`* interface for more details on making objects serializable.

5.3.6 CORBA considerations

Although the WorkArea facility can be used across Enterprise JavaBeans and the CORBA programming models, many composed data types cannot be successfully used across those boundaries. For example, if a SimpleSampleCompany instance is passed from the WebSphere environment into a CORBA environment, the CORBA application can retrieve the `SimpleSampleCompany` object encapsulated within a CORBA's `any` object from WorkArea, but it cannot extract the value from it. Likewise, an IDL-defined struct within a CORBA application and set into WorkArea is not readable by an application using the UserWorkArea class.

Applications can avoid this incompatibility by directly setting only primitive types, such as integers and strings, as values in WorkAreas, or by implementing complex values with structures designed to be compatible, like CORBA valuetypes. Also, CORBA Anys that contain either the `tk_null` or `tk_void` typecode can be set into WorkArea by using the CORBA interface, but the WorkArea specification cannot allow the J2EE implementation to return null on a lookup that retrieves these CORBA-set properties without incorrectly implying that there is no value set for the corresponding key. If a J2EE application tries to retrieve CORBA-set properties that are non-serializable or contain CORBA nulls or void references, the `com.ibm.websphere.workarea.IncompatibleValue` exception is thrown.

5.4 Configuring the WorkArea service in WebSphere Application Server

The OrderEntryServer configuration includes the necessary setup for the WorkArea service. There are very few settings to adjust.

5.4.1 Ensuring the WorkArea service is enabled in the Administrative Console

The WorkArea service is enabled by default. To make sure it has not been disabled, start the Administrative Console, and select the server from the **Application Servers** folder. From the Properties window, select the **Custom** tab. WorkArea Service should be listed in the window as shown in Figure 5-4.

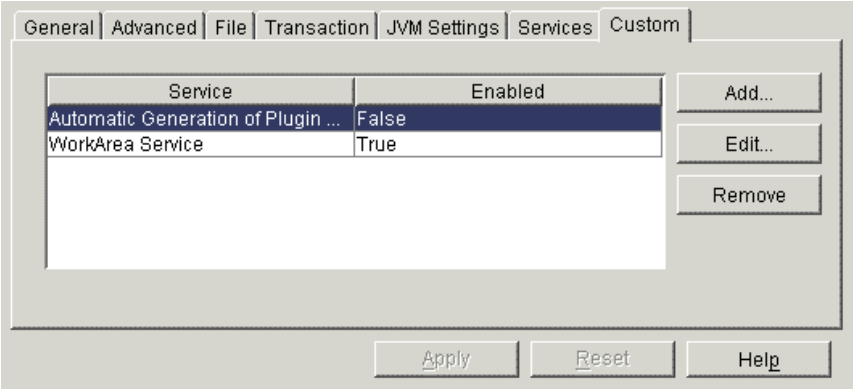


Figure 5-4 Administrative Console server properties window

5.4.2 Setting maximum send and receive size

The maximum size of WorkArea that can be sent from or received by the server can be set in the Administrative Console. From the **Custom** tab (Figure 5-4), select **WorkArea Service** and click the **Edit** button. From the pop-up window, select the **Custom** tab. This window has two values on it as shown in Figure 5-5. The default is 10000 bytes for both values.

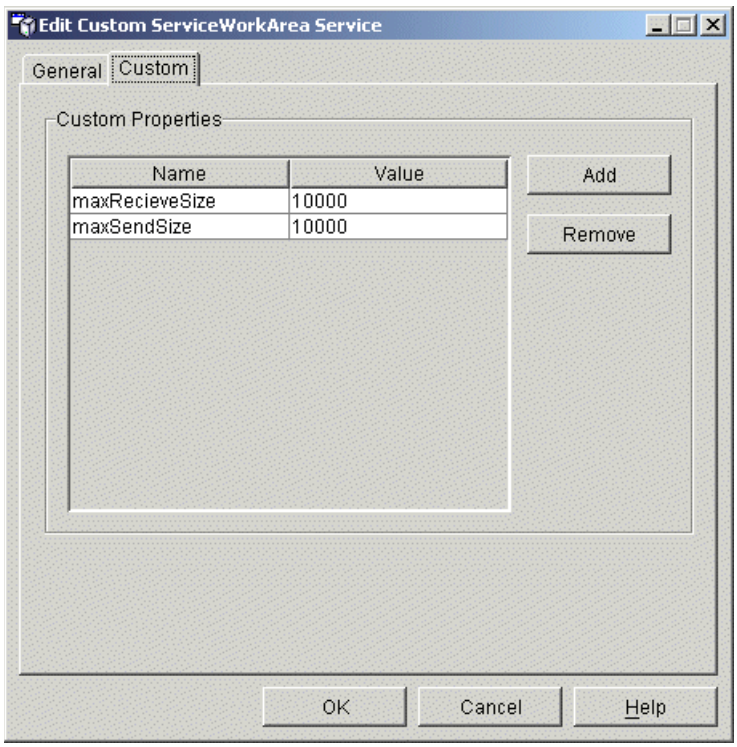


Figure 5-5 Edit WorkArea Service window

There are some rules and special values for two parameters of the WorkArea service:

- The maximum value for both settings is the maximum size of a Java integer type, 2 Gb.
- The smallest value is 1 byte, which means that no requests associated with WorkArea can enter or leave the system.

- A value of 0 means there is no limit to the size of WorkArea that can be sent or received.
- The default value is used if the value is specified improperly.

If you try to exceed the size limit of WorkArea, WebSphere Application Server throws the `org.omg.CORBA.IMP_LIMIT` exception during the request marshalling or unmarshalling.

5.4.3 Passing WorkArea through multiple WebSphere Application Servers

WorkArea can be started on a client and passed through multiple servers. WorkAreas are only passed through servers that have the WorkArea service installed. That means that WorkArea will not pass through an intermediary WebSphere Application Server Advanced server. However, WorkArea will pass through a WebSphere Application Server Enterprise Edition server that has disabled the WorkArea service.

6

The Internationalization service

The Internationalization service provides a mechanism to propagate locale and time zone information from clients to servers and between server components. This information can be used by server application components to customize results for the client. The transfer of the locale and time zone information is done transparently by the Internationalization service. The following topics are covered in this chapter:

- When to use the Internationalization service
- Using the Internationalization service
- Incorporating the Internationalization service in the example application
- Configure the Internationalization service on the WebSphere Application Server
- Enabling the tracing of the Internationalization service on the WebSphere Application Server

Note: Read the instructions in Appendix A, “Downloading and installing the OrderEntry application” on page 221, to understand the structure and environment settings used for the OrderEntry application.

© Copyright IBM Corp. 2002

6.1 The Internationalization service overview

In a distributed client-server environment, application processes may run on different machines configured to different locales corresponding to different cultural conventions. They may also be located across geographical boundaries. With the advent of Internet-based business computational models, like e-commerce, more and more clients and servers will operate in different locales and geographical regions.

The conventional solution for solving locale and time zone mismatch problems is to pass one or more extra parameters on all business methods necessary for conveying either the client's locale or time zone to the server. Although the technique is simple, it has serious limitations:

- Parameter lists become longer.
- Extra parameters may need to be added to methods that do not use the information just to pass the information through the call chain.
- Adding extra parameters to a deployed application is inherently error-prone.
- It may be impossible to add parameters to applications that are not easily modified, such as legacy applications.

6.1.1 Computers located in different locales

Client and server processes can run on computers that have a different locale setting. For example, a Spanish client may invoke a business method on an object that resides on an American server. Some business methods can be locale-sensitive. For example, a business method might return a sorted list of strings; the Spanish client will expect that list to be sorted according to the Spanish collating sequence, not in the server's English collating sequence. Since data retrieval and sorting procedures run on the server, the locale of the client must be available in order to perform a legitimate sort. A server may also have to return strings containing date, time, currency, or exception messages formatted according to the client's locale.

6.1.2 Computers located in different time zones

Client and server processes can execute in geographical locations having different time zones. For example, suppose a vendor makes the claim that "orders received before 2:00 PM will be processed by 5:00 PM the same day." The times given, of course, are in the time zone of the server that is processing the order. It is important to know the client's time zone in order to give customers in other time zones the correct times for same-day processing. Other time zone-sensitive operations include time stamping messages, order tracking, transaction completion time, and estimated shipment arrival times.

6.1.3 The Internationalization service solution

The WebSphere Enterprise Internationalization service solves the locale and time zone mismatch problems without the traditional limitations. The locale list and time zone are managed as a unit referred to as an internationalization context. The Internationalization service manages the distribution of internationalization context across the various components of Enterprise JavaBean applications, including Java client applications, Enterprise JavaBeans, JSPs, and servlets. Server-side components can use the internationalization context API to access distributed internationalization context and then localize computations according to the locale or time zone of client-side components.

The service associates an internationalization context with each thread of execution in an application. When a client-side program invokes a remote business method, the Internationalization service obtains the context associated with the current thread and attaches it to the outgoing request. At the server-side, the Internationalization service detaches the caller's context from the incoming request and associates it with the remote business method thread. The service propagates this context on subsequent remote business method invocations to pass the context of the original request down the call chain.

6.2 Internationalization context

Internationalization context is the information that is propagated and managed by the Internationalization service. There are two types of information managed by the service, and two types of context.

Important: We use the following terminology throughout the rest of the chapter:

- *Client* refers to an EJB application client deployed inside J2EE client container.
- *Invoking components* (may be either an EJB application client or the server application components, like servlets or EJBs) and *server application components* are used to identify two parties involved in the internationalization context propagation.

6.2.1 The information in the internationalization context

The internationalization context consists of an ordered list of `java.util.Locale` objects and a `java.util.SimpleTimeZone` object (`java.util.SimpleTimeZone` is a subclass of the abstract class `java.util.TimeZone`). Locale and SimpleTimeZone are defined in Sun's Java API.

- **Locale**: The `Locale` object is defined by a language and optional country and variant. An application can use `Locale` to properly format messages or sort string lists, for example, by passing it to locale-sensitive methods of other SDK objects, such as `java.text.NumberFormat`, `java.text.DateFormat`, or `java.text.Collator`.
- **TimeZone**: `TimeZone` represents a time zone offset from GMT. An application can use `TimeZone` to perform time-sensitive computations. For example, the constructors of `java.util.Calendar` accept a `TimeZone` parameter that configures the resultant `Calendar` objects to compute time according to the supplied time zone and daylight savings rules.

Refer to the Java SDK API documentation for a complete description of each type and the methods available.

Note: The Internationalization service currently only supports the `java.util.SimpleTimeZone` time zone type. Unsupported time zone types are *replaced* with the default `SimpleTimeZone` of the JVM when supplied to service API methods.

6.2.2 Types of context

There are two types of context:

- Caller context
- Invocation context

Both consist of an ordered locale list and a time zone.

Caller context

Caller context is the Internationalization context that server application components (EJBs, servlets, JSPs) receive on incoming requests originating from any J2EE application component. Note that such requests may immediately originate from the server components, such as EJBs and servlets, as well as J2EE application and HTTP clients. The server application components can access locale and time zone information through an object that implements the *com.ibm.websphere.i18n.context.Internationalization* interface.

Caller context received by the server application components will be available to all server application components that are associated with the current thread of execution, provided the Internationalization service is enabled on the containing application server. Caller context may *not* be changed by the server application components.

Invocation context

Invocation context is the context under which a component or business method executes. On business method invocations, it is the context that propagates from the invoking components to the corresponding server application components that implement these methods. Invocation context may be accessed by using the *com.ibm.websphere.i18n.context.InvocationInternationalization* interface of the Internationalization Service API. Invocation context is managed according to the policies defined for the client and server application components:

- Client-side Internationalization (CSI) policy for the client application components
- Server-side Internationalization (SSI) policy for the server application components

In particular, these polices specify API access restrictions, how invocation context propagates on remote requests, and the context under which a target request executes.

The server application components can use both *Internationalization* and *InvocationInternationalization* interfaces. However, in the current release, according to the SSI policy, the caller context is used as the invocation context for the server application components. For more information on the SSI policy, see “Server-side Internationalization (SSI) policy” on page 111.

6.2.3 The Internationalization Service API

Applications use the service to access and manage internationalization context. Three interfaces are provided by the `com.ibm.websphere.i18n.context` package for this purpose:

- *com.ibm.websphere.i18n.context.UserInternationalization*
- *com.ibm.websphere.i18n.context.Internationalization*
- *com.ibm.websphere.i18n.context.InvocationInternationalization*

The UserInternationalization interface

The *UserInternationalization* interface provides a factory for obtaining the internationalization API objects. These objects give access to the desired type of an internationalization context (caller or invocation contexts). The interface is shown in Example 6-1.

Example 6-1 The Internationalization service interface

```
public interface UserInternationalization {
  public Internationalization getCallerInternationalization();
  public InvocationInternationalization getInvocationInternationalization();
}
```

The *UserInternationalization* interface defines two methods:

- *Internationalization getCallerInternationalization()*

  Returns an object implementing the *Internationalization* interface. This interface allows read-only access to the caller context. If the service is disabled, the method throws a `java.lang.IllegalStateException` exception.

- *InvocationInternationalization getInvocationInternationalization()*

  Returns an object implementing the *InvocationInternationalization* interface. This interface allows read and write access to the invocation context according to the internationalization context management policies. In the current release, only an application client may change the invocation context. If the service is disabled, the method throws a `java.lang.IllegalStateException` exception.

The Internationalization interface

The *Internationalization* interface allows read-only access to the internationalization context. Use this interface to access the caller context within server-side application components. Example 6-2 shows the *Internationalization* interface.

Example 6-2 The Internationalization interface

```
public interface Internationalization {
  public java.util.Locale[] getLocales();
  public java.util.Locale getLocale();
  public java.util.TimeZone getTimeZone();
}
```

The *Internationalization* interface provides three methods for read-only access to the internationalization context elements:

- *java.util.Locale [] getLocales()*

  Returns the array of locales associated with the current thread. If the array is null, the method returns an array of locales of length one containing the default locale of the process associated with the execution of this method. The method is useful if the first locale in the array is not recognized or not supported by the server. A different locale from the array could be selected.

- *java.util.Locale getLocale()*

  Returns the first element from the array of locales associated with the current thread. If the array of locales is null, the method returns the default locale of the process associated with the execution of this method.

- *java.util.TimeZone getTimeZone()*

  Returns the `SimpleTimeZone` object associated with the current thread. If the time zone is null, the method returns the default time zone of the process associated with the execution of this method.

Note: Due to the inheritance, caller and invocation contexts can be accessed using the *Internationalization* interface.

The InvocationInternationalization interface

The *InvocationInternationalization* interface allows read and write access to the invocation context. Use this interface to read or modify the invocation context. But remember, client and server application component access to the invocation context is governed by the CSI or SSI policy respectively. Example 6-3 shows the *InvocationInternationalization* interface.

Example 6-3 The InvocationInternationalization interface

```
public interface InvocationInternationalization extends Internationalization {
  public void setLocales(java.util.Locale [] locales);
  public void setLocale(java.util.Locale locale);
  public void setTimeZone(java.util.TimeZone timeZone);
  public void setTimeZone(String timeZoneId);
}
```

Since the *InvocationInternationalization* interface extends the *Internationalization* interface, all methods from the *Internationalization* interface are available in the *InvocationInternationalization* interface. In addition, the *InvocationInternationalization* interface defines four more methods:

- *void setLocales(java.util.Locale [])*

  This method sets the array of locales, associated with the current thread, to the supplied array of locales. The supplied locale array can be null or have length greater than or equal to zero. When the supplied locale array is null or has length of zero, the Internationalization service sets the array of locales to an array length of one containing the default locale of the process associated with the execution of this method. Null entries can exist within the supplied array of locale. However, on the remote method invocations, the Internationalization service substitutes all null elements with the default locale of the process associated with the execution of this method.

- *void setLocale(java.util.Locale)*

  This method sets the array of locales, associated with the current thread, to an array of length one containing the supplied locale. If the supplied locale is null, the service sets the array of locales to an array of length one containing the default locale of the process associated with the execution of this method.

- *void setTimeZone(java.util.TimeZone)*

  This method sets the invocation time zone, associated with the current thread, to the supplied time zone. If the supplied time zone is null or not an instance or subclass of `java.util.SimpleTimeZone`, the service sets the time zone to the default time zone of the process associated with the execution of this method.

- *void setTimeZone(String)*

  This method sets the time zone, associated with the current thread, to `SimpleTimeZone` with the supplied ID. The general format for time zone ID is either “country/city” or “GMT[+|-]hh[[:]mm]”. For example, the time zone ID for the U.S. Pacific Time zone is “America/Los_Angeles” or “GMT-08:00”. If the supplied time zone ID is null or unsupported, the Internationalization service sets the time zone to a time zone having an

ID of “GMT” and the default offset of the process is associated with the execution of this method. The use of three-letter time zone IDs other than “GMT” is deprecated. A list of supported time zone IDs can be obtained using the *java.util.TimeZone.getAvailableIds()* method. See the Java SDK API documentation for more information about time zone IDs.

6.2.4 Internationalization context management

The Internationalization service transparently propagates locales and time zones across J2EE clients, Enterprise JavaBeans, JSPs, and servlets. The context management policies of the service determine how the internationalization context is propagated.

The following context management policies are applied depending on whether an application component is contained on the client side or the server side.

Client-side Internationalization (CSI) policy

Only EJB client application components deployed within J2EE client containers are subject to the Client-side Internationalization policy.

In CSI, client applications can get and set the invocation context locale and time zone elements associated with the current thread using the `InvocationInternationalization` interface. Invocation context elements set by the client persist until they are set again or until the application exits. Clients may also get the caller context elements associated with the current thread. In a client application, caller context is *always* the default locale and time zone of the client’s JVM.

On the remote method calls to EJBs, originated from a J2EE client application, the Internationalization service propagates all invocation context elements associated with the current thread to the outgoing request. If an invocation context element was set using the `InvocationInternationalization` interface, the service propagates the specified element. Otherwise, the service propagates the default values that are associated with the client’s JVM at that time.

Server-side Internationalization (SSI) policy

EJB application components deployed within J2EE server-side containers are subject to the Server-side Internationalization policy. SSI specifies how internationalization context propagates across the Web and Enterprise JavaBean container boundaries and defines access privileges to internationalization context within contained application components.

In SSI, server-side application components always run under the caller’s internationalization context. The service associates the context of an incoming business method request to the thread on which that method will execute. The incoming caller context is associated as both the caller and invocation context for that method and persists until the method returns. This policy is applied directly to Enterprise JavaBean methods.

SSI is defined in a different way for the servlet methods (such as *doGet* or *doPost*) because they are invoked using Web browser clients. On incoming servlet requests from a Web browser client, the Internationalization service associates the list of locales, propagated by the browser within the HTTP header, and the *server’s* default time zone to the thread on which the servlet service method will execute. The client’s time zone is not propagated from the browser to the server.

On the remote method calls to EJBs, originated from either a servlet or another EJB, the Internationalization service propagates all invocation context elements associated with the current thread on the outgoing requests. If an invocation context element was set using the API, the Internationalization service propagates the specified element. Otherwise, the

Internationalization service propagates the default element associated with the invoking the component's JVM at that time. This is identical to CSI, except that in SSI the invocation context is always that of the caller. This management policy could be referred to as "RunAsCaller/DefaultToServer".

6.3 Using the Internationalization service in the example application

With the Internationalization service enabled on the server, the HTTP client's context is propagated to the server when the client accesses `OrderEntryServlet`. This context is then available to all components of the OrderEntry application. To use the internationalization context, we must:

- Bind to the Internationalization service.
- Use the *`UserInternationalization`* interface to retrieve API objects that afford access to the desired Internationalization context types.
- Access the desired context type.

Once the context is accessed, we can use it with time zone and locale-sensitive operations.

6.3.1 Binding to the Internationalization service

Example 6-4 shows the `ItemsViewBean` constructor that binds to the Internationalization service using JNDI. Then the interfaces to the caller and invocation contexts are retrieved.

Example 6-4 Binding to the Internationalization service

```
import com.ibm.websphere.i18n.context.*;
import javax.naming.*;
...
private UserInternationalization i18nService;
private Internationalization callerI18N;
private InvocationInternationalization invI18N;
...
public ItemsViewBean() {
   InitialContext jndi = null;

   // Bind to Internationalization facility
   try {
      jndi = new InitialContext();
      i18nService = (UserInternationalization)
                     jndi.lookup("java:comp/websphere/UserInternationalization");
   }
   // This exception indicates a problem binding to the facility
   catch( NamingException e ) { e.printStackTrace(); }}

   // Bind to the desired interfaces
   try
   {
      callerI18N = i18nService.getCallerInternationalization();
      invI18N = i18nService.getInvocationInternationalization();
   }
   // This exception indicates the service is disabled
   catch (IllegalStateException e)
   {
```

```
        e.printStackTrace();
    }
}
```

Tip: Internationalization context API references need to be resolved only once over the life cycle of any application component. When developing server-side application components, like servlets and Enterprise JavaBeans, bind to the Internationalization service within the initialization methods of these components, for example, within the *init()* method of servlets or within the *ejbSetContext()* method of EJBs.

6.3.2 Retrieving and using the caller internationalization context

Once we bind to the Internationalization service, we can retrieve the caller internationalization context and use it. Example 6-5 shows a method in the `ItemsViewBean` that retrieves the caller context and uses it to format the currency amount according to the invoking component's locale. Note that if currency exchange rate must be calculated, we would have to call a method to do that in the place indicated. The method to calculate exchange rate is beyond the scope of this example. However, note that because the method executes on a WebSphere Application Server Enterprise Edition server with the Internationalization service enabled, it also has access to the caller locale. There is no need to pass locale as a parameter to the method.

Example 6-5 Retrieving and using the caller internationalization context

```
import com.ibm.websphere.i18n.context.*;
import java.text.NumberFormat;
...
private Internationalization callerI18N;
private float orderTotal = 0;
...
public String getOrderTotalForDisplay()
{

    java.util.Locale callerLocale;
    java.text.NumberFormat nf;
    float currencyAmount;

    callerLocale = callerI18N.getLocale();
    nf = java.text.NumberFormat.getCurrencyInstance(callerLocale);

    // This is where the exchange rate should be calculated
    currencyAmount = calulateExchangeRate( orderTotal );
    return nf.format( currencyAmount );
}
```

Important: Some browsers may send a locale that specifies a language code but no country code. Server application components accessed from a client browser may need to use means other than the Internationalization service to determine the correct country code for the client. This could be accomplished by asking the HTTP client to specify their country if the country code was not received with the locale.

6.3.3 Retrieving and using the invocation internationalization context

Example 6-6 shows a method that retrieves the time and formats it according to the invocation internationalization context. Note that in the example application, the invoking component requests are made to a servlet from an HTTP browser client, so the invocation and caller time zone are actually the JVM time zone of the server.

Example 6-6 Retrieving and using the invocation internationalization context

```
import com.ibm.websphere.i18n.context.*;
import java.util.*;
...
private InvocationInternationalization invI18N;
...
public String getTime()
{

   TimeZone timeZone;
   Locale locale;

   // Get the time from the invocation
   timeZone = invI18N.getTimeZone();
   // Get the locale (for formatting) from the invocation
   locale = invI18N.getLocale();

   // Get the current date and time for the specified time zone
   GregorianCalendar cal = new GregorianCalendar(timeZone, locale);
   Date date = cal.getTime();
   // Format the date and time for the locale
   DateFormat df =
      DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL, locale);
   return df.format(date);
}
```

Retrieving and using the invocation context is similar to retrieving and using the caller context. The main difference is that the invocation context may be set and changed by the client while the caller context is set by the Internationalization service and may not be changed.

6.4 Enabling or disabling the Internationalization service

In order to use the Internationalization service, it must be first enabled on WebSphere Application Server Enterprise Edition. If the client is an EJB Java application client, the service must also be enabled on the client. HTTP browser clients do not require enabling of internationalization; this support is provided by the browser.

6.4.1 Enabling internationalization context for an application server

Any Enterprise JavaBean, JSP, or servlet can use the internationalization context once the Internationalization service is enabled on the application server.

The Internationalization service is enabled within application servers using the **Custom** tab. The **Custom** tab lists all Enterprise Services available to a particular application server and provides access to the configuration properties of each.

Perform the following steps to enable internationalization context within an application server:

1. In the WebSphere Advanced Administrative Console window, expand **WebSphere Administrative Domain-> Nodes-> *node*-> Application Servers-> *application_server***.

   The properties related to the selected application server are displayed in the right-hand pane.

   Note: *node* represents the node on which the application server is located, and *application_server* represents the name of the application server for which you are enabling the service.

2. In the right-hand pane, click the **Custom** tab (see Figure 6-1).
   - If the Custom tab does not list the Internationalization Service entry, you must add a new entry for Internationalization Service. Go to step 3.
   - If the Custom tab lists the Internationalization Service entry, you can view and edit the Internationalization Service properties. Go to step 4.

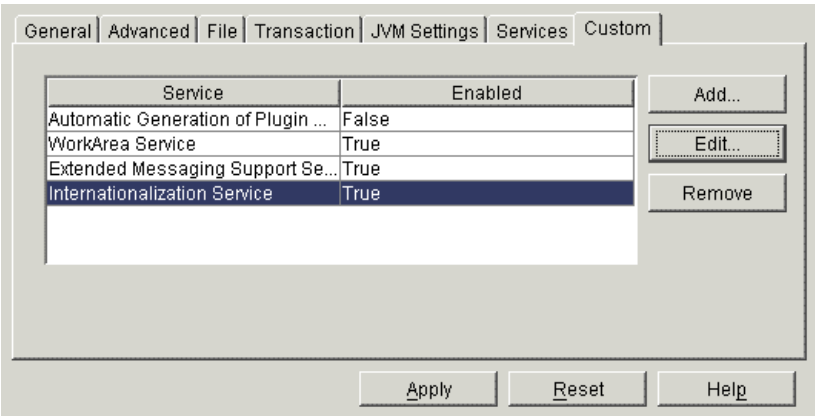


Figure 6-1 Custom tab with Internationalization Service selected

3. Add a new Internationalization Service entry to the Custom tab:
   a. On the Custom tab, click **Add**. The Add Custom Service dialog is displayed.
   b. In the Add Custom Service dialog, enter the properties as shown in Figure 6-2.
   c. Click **OK**. The Add Custom Service dialog closes.
   d. Click **Apply**. Your changes are saved.
4. Verify or edit the Internationalization Service properties:
   a. On the **Custom** tab, click **Edit**. The Edit Custom Service dialog is displayed.
   b. Click the **General** tab.
   c. In the Classname field, verify or change the following entry (see Figure 6-2):

      `com.ibm.ws.i18n.context.ServiceInit`

   d. Select the **Enabled** check box, if not already selected.
   e. Click **OK**. The Edit Custom Service dialog closes.
   f. Click **Apply**. Your changes are saved.

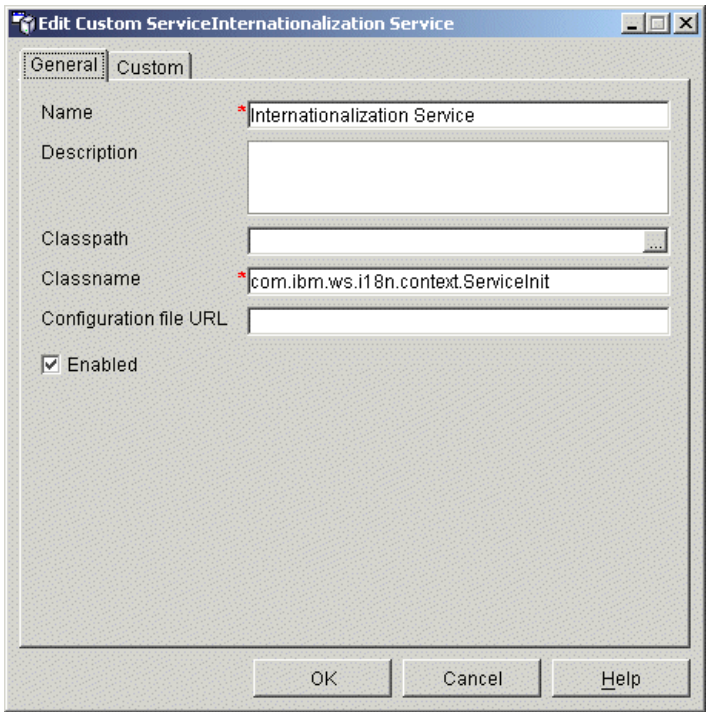


Figure 6-2 Edit Custom Service dialog

These settings cause the Internationalization service to initialize when starting or restarting the corresponding application server.

6.4.2 Enabling internationalization context within EJB Java application clients

To enable the Internationalization service for EJB Java application clients, the i18nctx.jar file must appear in the CLASSPATH constructed by the **`launchClient`** utility. When invoking a client application, **`launchClient`** correctly configures the CLASSPATH to include the i18nctx.jar file. Then it initializes the service for use within that application. See the WebSphere InfoCenter documentation “Launching Java application clients in the J2EE application client container” for more information about using the **`launchClient`** utility.

6.4.3 Configuring the programming environment

The `java.util` and `com.ibm.websphere.i18n.context` packages contain all classes necessary to use the Internationalization service within an EJB application.

Classes that are specific to the Internationalization service reside in the %WAS_HOME%\lib\i18nctx.jar file. Be sure to add the i18nctx.jar file to the CLASSPATH when compiling application components that import Internationalization service classes.

6.5 Beyond the example application

This section provides information for advanced topics about the Internationalization service. It includes information about:

- Setting invocation locale and time zone
- Tracing the Internationalization service functions

6.5.1 Setting invocation locale and time zone

The client may set the invocation locale and time zone using the *set* methods provided by the *InvocationInternationalization* interface. In Example 6-7, locale (en, GB) and simple time zone (GMT) transparently propagate on remote calls to server-side components. Server-side application components, such as a servlet, JSP, or EJB may use the Internationalization service to obtain the client locale and time zone.

Example 6-7 Setting the invocation context elements

```
import com.ibm.websphere.i18n.context.*;
import java.util.*;

   ...
   private UserInternationalization i18nService;
   private InvocationInternationalization invocationI18n;
   ...

   try {
      InitialContext jndi = new InitialContext();
      i18nService = (UserInternationalization)
                     jndi.lookup("java:comp/websphere/UserInternationalization");
      invocationI18N = i18nService.getInvocationInternationalization();
   }
   // An exception indicates the service is disabled or could not be found
   catch (Exception e)
   {
      e.printStackTrace();
   }

   // Set the invocation locale to en,GB
   Locale localeToPropagate = new Locale("en", "GB");
   invocationI18n.setLocale(localeToPropagate);

   // Set the invocation time zone to GMT
   SimpleTimeZone timeZoneToPropagate = (SimpleTimeZone)SimpleTimeZone.getTimeZone("GMT");
   invocationI18n.setTimeZone(timeZoneToPropagate);

   // Call remote method
   ...
```

6.5.2 Tracing the Internationalization service function

The trace service function can be enabled on the application server to emit trace statements regarding the Internationalization service function.

To enable the Internationalization service trace, follow these steps:

1. Start the application server you want to trace.
2. In the WebSphere Advanced Administrative Console window, expand **WebSphere Administrative Domain-> Nodes->** ***node*****-> Application Servers->** ***application_server***.

   The properties related to the selected application server are displayed in the right-hand pane.

Note: *node* represents the node on which the application server is located, and *application_server* represents the name of the application server on which the trace will be enabled.

3. In the right-hand pane of the WebSphere Advanced Administrative Console window, select the **Services** tab.
4. On the Services tab, select **Trace Service** from the list of services (see Figure 6-3).

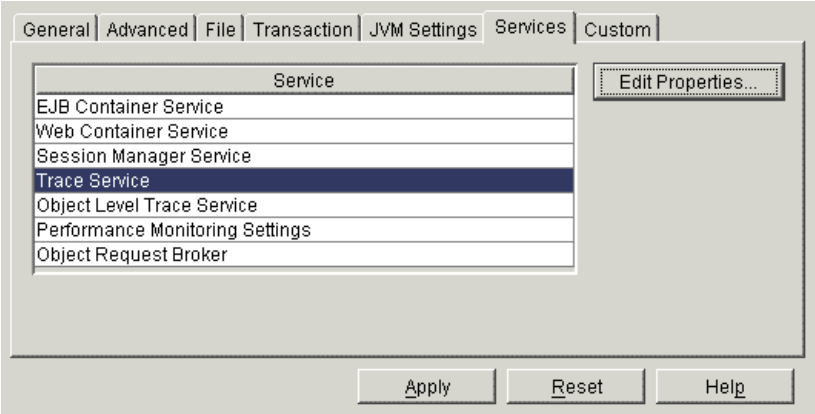


Figure 6-3 Services tab

5. Click **Edit Properties**. The Trace Service dialog is displayed.
6. In the Trace Service dialog, click the button to the right of the Trace specification field to open the Trace dialog (see Figure 6-4).

Note: In the Trace Service dialog box, you may also specify an output file to log application server trace messages to, including those generated by the Internationalization service.

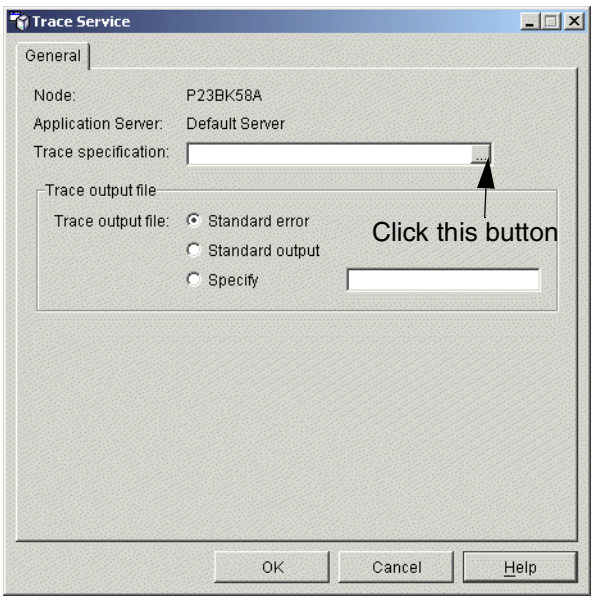


Figure 6-4 Trace Service dialog box

7. In the Trace dialog, expand **Components-> com-> ibm** to see II18nContextImpl in the list (see Figure 6-5).
8. Right-click **II18nContextImpl** and select **All** from the pop-up menu.

   The icon to the left of II18nContextImpl will change from gray to blue/yellow/red to indicate the trace has been selected (see Figure 6-5).

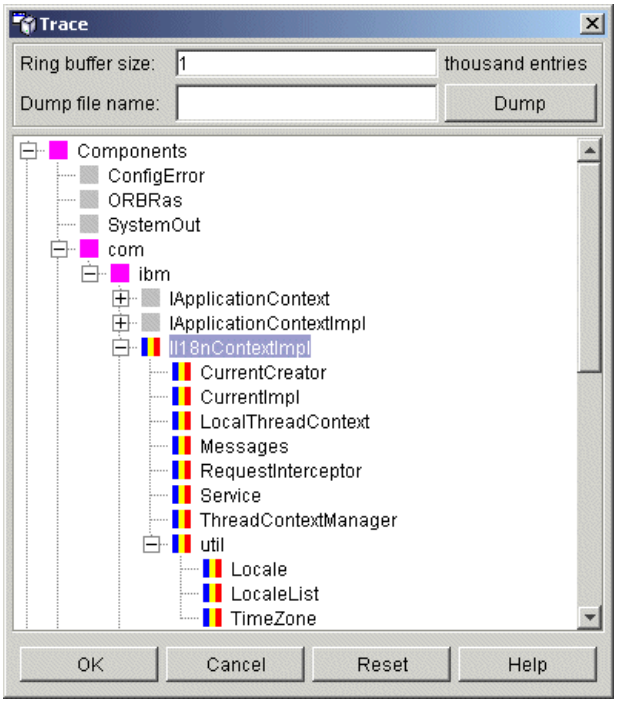


Figure 6-5 Trace dialog with II18nContextImpl trace selected

9. Repeat steps 7 and 8 for the `com.ibm.ws.i18n.context` package (see Figure 6-6).

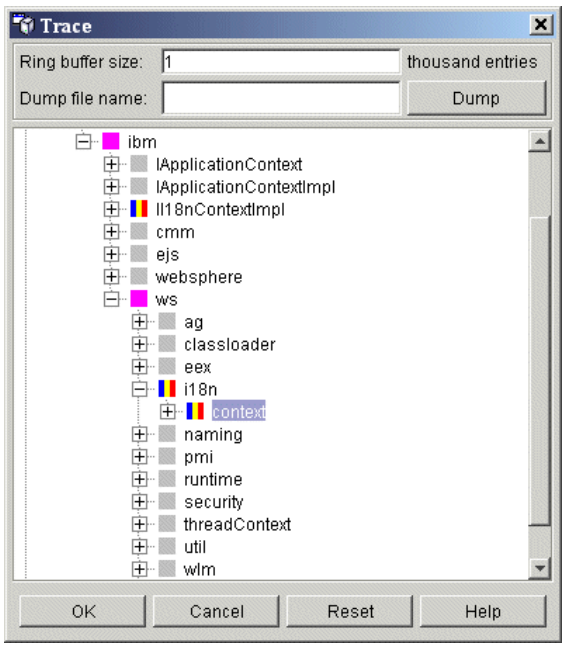


Figure 6-6 Trace dialog with com.ibm.ws.i18n.context trace selected

10. Click **OK** in the Trace dialog. The Trace dialog closes.
11. Click **OK** in the Trace Service dialog. The Trace Service dialog closes.
12. Click **Apply** on the Services tab. The changes are applied.

> **Note:** Clicking Apply enables the trace and makes the trace persistent across server shutdown and restart.
>
> To disable the tracing function, follow the above steps, selecting **None** instead of All from the Trace window pop-up menu. The icons in the Trace dialog will return to gray to indicate the trace is disabled.

Restart the application server. Messages tracing the function of the Internationalization service are output to the file specified in the Trace Service dialog.

7

The ActiveX bridge service

This chapter discusses the ActiveX to EJB bridge, a technology that enables Component Object Model (COM) applications and services to connect to EJBs in WebSphere and leverage other J2EE services. It covers the components of the bridge and discusses installation and configuration issues. It also uses a sample COM client that has been modified to take advantage of the bridge technology, using code samples to demonstrate some programming guidelines. There is also a reference to the classes and methods provided by the bridge at the end of the chapter.

This chapter covers the following topics:

- The ActiveX bridge components
- Supported languages and platforms
- Enabling COM technologies to use the ActiveX bridge
  - Configuring the client container and client side EAR file
  - Initializing the JVM
  - Launching the client container
- Developing and Debugging
  - Using the bridge from a development environment
  - Instantiating and Invoking Java objects through the bridge
  - Best practice programming guidelines
  - Error handling
- Class reference

Note: To understand the structure of the OrderEntry application and to run it, read the instructions in Appendix A, “Downloading and installing the OrderEntry application” on page 221.

© Copyright IBM Corp. 2002

7.1 Introduction

Many organizations have made considerable investments in COM-based technologies such as Visual Basic, Visual C++, and Active Server Pages (ASP). To save re-developing these services as J2EE components, the ActiveX bridge offers the ability to attach both ActiveX clients and ActiveX servers to EJBs by initializing a JVM within the ActiveX process. The bridge enables access to all Java classes and objects within the JVM and remains attached to the process until that process terminates.

The suggested approach is to take advantage of the WebSphere Application Client architecture by launching a J2EE client container (CC) in the attached JVM. This provides the COM process with simple access to J2EE services and requires minimal configuration since the EJB stubs are already contained in the Enterprise Archive file (EAR) file. Figure 7-1 shows the position of the bridge and demonstrates the option of connecting to the EJB container via a client container or straight from the JVM.

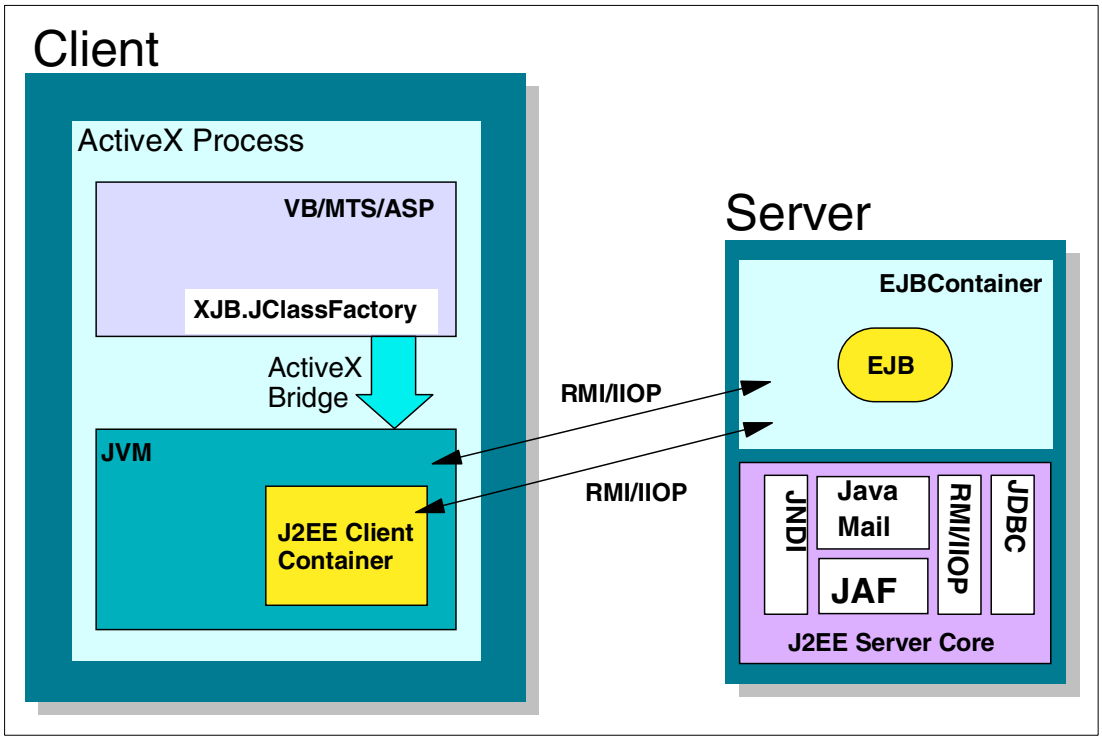


Figure 7-1 Positioning the ActiveX bridge

The ActiveX bridge consists of a C++ DLL that calls some Java code using Java Native Interface (JNI) (Figure 7-2). The Java code then invokes the services of either a J2EE client container or a Java proxy, which in turn makes a call to a Java ORB.

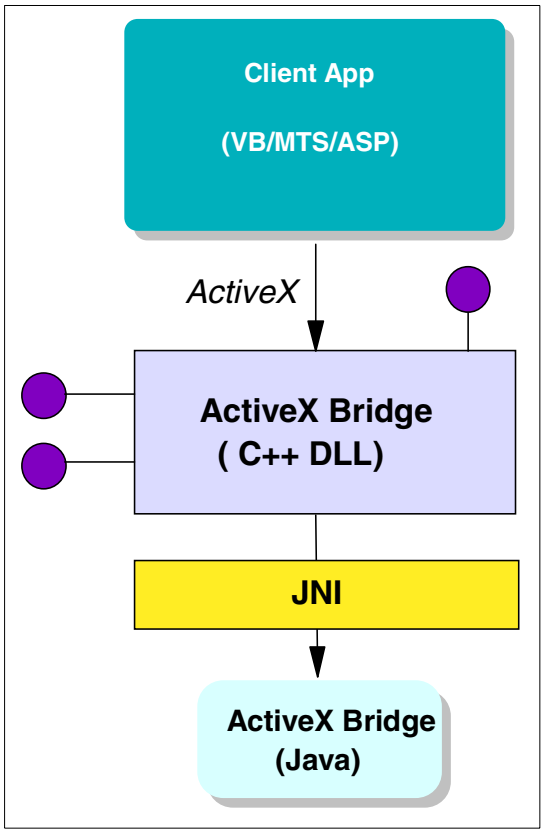


Figure 7-2 Components of the ActiveX bridge

The bridge also enables connection to J2EE APIs such as Java Naming and Directory Interface (JNDI), Java Database Connectivity (JDBC), Java Messaging Service (JMS), and Java Mail and URLs. The bridge supports both workload management and security.

All exceptions thrown in Java code are encapsulated and rethrown as a COM error, from which the ActiveX program can determine the actual Java exceptions.

The ActiveX to EJB bridge supports both free-threaded and apartment-threaded access and implements the Free Threaded Marshaler to work in a hybrid environment such as Active Server Pages.

To enable your ActiveX process to use the ActiveX bridge functionality the WebSphere Application Client, supplied with WebSphere Application Server Advanced Edition, must be installed on the client. The ActiveX bridge enterprise service is then installed on top of the application client. Now, when the client creates a JVM, it will have access to the ActiveX bridge functionality.

7.1.1 Supported languages and platforms

ActiveX programs can be written in Visual Basic (VB), VBScript, ASP, Delphi, PowerBuilder, and other COM supporting languages. Although the ActiveX to EJB bridge should work from any automation component, the only formally supported languages are Visual Basic, VBScript, and ASP. All the examples in this chapter will deal with Visual Basic and ASP.

An issue with being unable to invoke case-sensitive Java methods prevents an easy implementation of a PowerBuilder solution. See 7.5.4, “Invoking Java methods” on page 137. The ActiveX bridge is supported on both Windows 2000 and Windows NT.

7.2 The Visual Basic OrderEntry client application

Before digging into the details of the actual code, let's look at the application design issues.

7.2.1 Designing an ActiveX client program

A good software design promotes component reusability and its independence from other components in the system. Many designs exist today, but one of them has emerged as the de facto standard – the Model/View/Controller (MVC) pattern.

This is a well known recipe for writing software that provides simultaneously different views on the same set of data. It provides a clear way of encapsulating a different part of an application in the different components. The roles (participants) in the MVC pattern are (see Figure 7-3):

- The *Model* encapsulates application state (data) and operations that modify that state. A model is meant to serve as a computational approximation or abstraction of some real world process or system. For example, the Model of a shopping cart of an e-commerce application contains the identity and number of purchased items and offers operations to add and remove items from the shopping cart.
- The *View* is responsible for presenting (rendering) the information contained in the Model in a particular output modality. For example, a graphical View may render a shopping cart as a list showing the items and their number, plus a "remove" button next to each item.

  A View is an observer of the Model (and therefore also depends on the Model). At initialization time, the View registers with the Model to express its interest in any changes of the Model's state. Whenever the Model changes states, it notifies the View by sending it an EventObject that contains the state change. The View then updates its rendering.
- The *Controller* accepts input from the user in a particular modality, interprets the input (the interpretation may depend on the View), and invokes the appropriate operation of the Model. For example, when the Controller detects a mouse click event on the Remove button of an item, it invokes the remove operation on that item. Any state changes that this operation effects on the Model are sent, by the Model, to the registered Views via events and EventObjects.

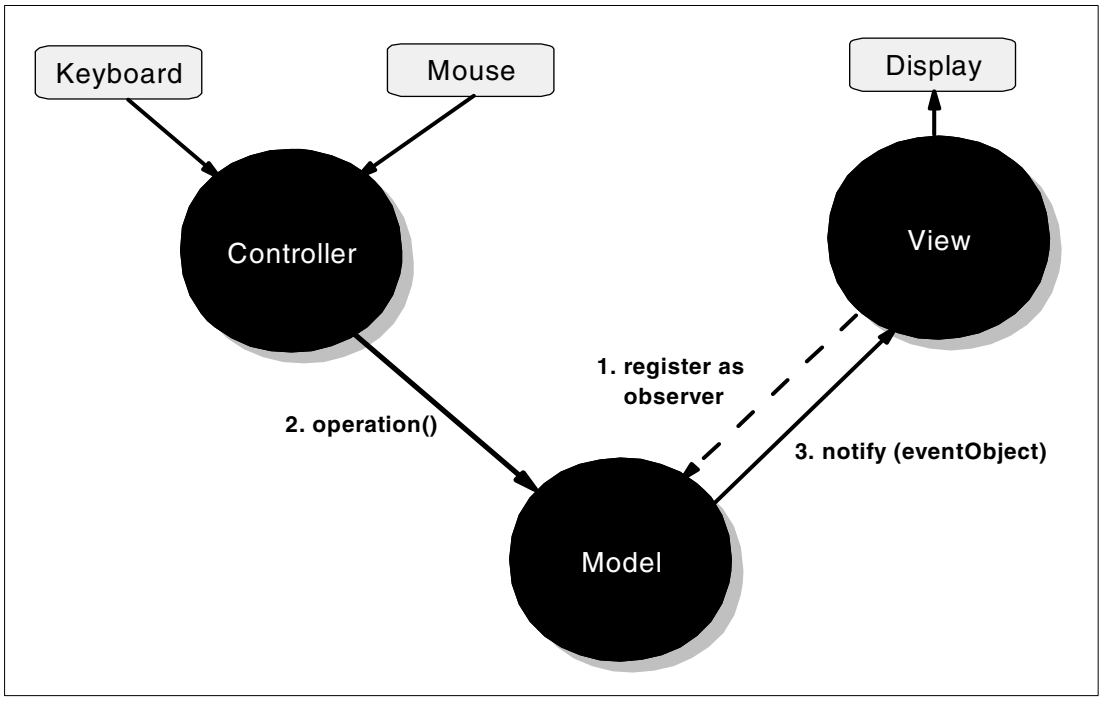


Figure 7-3 MVC pattern

The MVC pattern enables the combination of a single Model with multiple Views and Controllers. The MVC pattern ensures that the Views are kept synchronized as can be seen from the following example. One of the Controllers recognizes a valid command from the user's input. The Controller invokes the corresponding method on the Model. The Model verifies that the operation is compatible with the current state, executes it, and changes the state correspondingly. The state change triggers an event that the Model multicasts to each View.

To support multiple Views and Controllers, dependencies must be kept minimal. Another precondition to support multiple Views and Controllers is to keep interactions minimal. In particular, a Controller must never directly affect the rendering of its associated View. Instead a user input must make a complete round trip through the Model before its effects become visible in the View. This rule guarantees that a state change updates all Views and the Views remain synchronized. Often implementations with a single Controller violate this rule because of a lack of thought: "I already know that this state change will occur, and therefore, do not need wait for the Model to tell me about it". This is wrong for two reasons:

- The Model can veto the operation for some reason. The operation will not occur.
- Other Controllers concurrently invoke operations on the Model. Some other operation can slip in between, which fundamentally changes the rendering and makes any assumptions about it invalid.

We applied this pattern to the design of the ActiveX client for the OrderEntry application. The ActiveX client plays two roles: View and Controller. Referenced EJBs (hosted by WebSphere Application Server) are the Model that represents the data and operations on that data. Such distribution of the three components in the MVC pattern is called *server-centric placement*.

7.2.2 ActiveX client interface

The client is designed to represent the application used by call center agents to place orders for customers. It uses the OrderEntry Application EAR file both on the client and the server. The EAR file has been modified to include the client.jar file (see 7.8, “Modifying the EAR file for the ActiveX J2EE client container” on page 148).

Customers can be selected by entering their customer ID or performing a customer search to list all the customers in the database. Once the customer has been chosen, the items for order are selected in a similar manner, either by Item ID or following an Item search that displays all the Items in the database. Choosing the quantity and pressing the Tab key highlights the Add Item button. Once enough items are selected, the order can be processed. This returns the order number. Figure 7-4 shows an example of the clients main window.

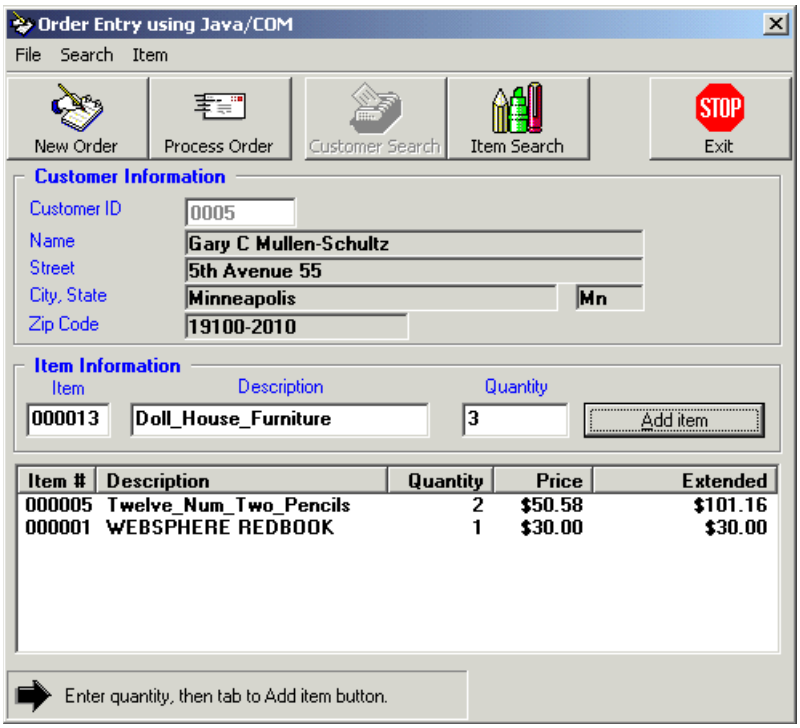


Figure 7-4 The Order Entry Client that uses the ActiveX to EJB bridge

7.2.3 Installing the client application

After installing the WebSphere Application Client and ActiveX bridge service, place the %OE_HOME%\ActiveX\OrderEntry.ear file in the directory %WAS_HOME%\WSsamples\Client\J2EE\OrderEntry.

Here *%WAS_HOME%* (for example, C:\IBM\WebSphere\AppClient) is the environment property set up when installing the WebSphere Application Client or when either the `launchClientXJB` or `setupCmdLineXJB` batch files are run (see 7.4.6, “Launching a client container in an ActiveX process” on page 131). The COM client uses this path to find the EAR file. If there is a need to place the file somewhere else, the `EARFileName` parameter, in the `ActiveX_Config` module, in the VB Order Entry Client must be changed.

Run OrderEntryClient.exe using the **`launchClientXJB`** command from a command prompt:

```
%WAS_HOME%\Enterprise\bin\launchClientXJB %OE_HOME%\ActiveX\OrderEntryClient.exe
```

The first window to appear is the options window. Enter information in the Bootstrap Host and Port fields (900 is the default port for WebSphere Application Server). The client application will now initialize the JVM and client container and display the main window.

7.2.4 The client application components

The Visual Basic client code is in OrderEntryClient VB Project. The application was originally purely COM based, but has been enabled to use the ActiveX to EJB bridge by adding some helper classes and adapting the code. The client application can be run within Visual Basic IDE by issuing the command:

```
launchClientXJB %OE_HOME%\ActiveX\Source\OrderEntryClient.vbp
```

This launches the Visual Basic IDE in an ActiveX enabled process. The client can now be run in debug mode. If the `launchClientXJB` batch file is not run, the client cannot initialize the JVM.

The following sections offer a brief description of the major components of the Visual Basic Order Entry Client.

ActiveX_XJBHelpers

This module should be added to any Visual Basic application that is to be modified to take advantage of the ActiveX to EJB bridge. The generic methods are used to initialize the JVM and client container and obtain references to EJB Home Interfaces:

- **XJBInit**: Creates and initializes the JVM
- **XJBGetJ2EEClientContainer**: Gets the J2EE Client Container object
- **XJBGetEJBHome**: Gets the Home EJB Object
- **XJBPutProperty**: Adds a string key/value to a Java Properties object
- **XJBGetProperty**: Gets a string key/value to a Java Properties object
- **XJBGetEnvEntry**: Gets an environment entry from the J2EE client container namespace

modMain

The *main* method of this module is the first method to be called when the application is run. It is responsible for initializing the global `g_Config` object and displaying the options window.

Class modules

This section presents the custom-built class modules. They represent the Visual Basic part of the application.

ActiveX_EJBHomeRef

This class is used to hold a reference to all the EJB Home Interfaces. It uses *XJBGetEJBHome* in the ActiveX_XJBHelpers module during initialization. It is referenced via the `g_Config` object, which ensures that it is initialized.

ActiveX_Config

This class has three roles:

- Holds the `XJB.JClassFactory` object throughout the life of the application. The only instance of this class is created when the application is launched. It is public so other classes can reference `XJB.JClassFactory` and, therefore, the JVM by calling `g_Config.XJB`. The JVM and CC are initialized when `g_Config` is first called. This happens just after the main window is displayed.
- Holds the reference to the initialized `ActiveX_EJBHomeRef` object. This object is first initialized when `g_Config.EJBHomeRef` is first called. This happens just after the main window is displayed.

- In this example, it stores configuration parameters, such as the naming factory class, and the server address and port number that are obtained from the options window. It also stores all the parameters that the application needs to initialize and extract the client EAR file.

COrderClerk

This class provides the main functionality for the client. It holds the Order Entry Clerk EJB reference and uses it to retrieve a list of customers and items when the user selects a search button. It also uses Customer and Item EJB home interfaces retrieved from the `ActiveX_EJBHomeRef` module to invoke the *findByPrimaryKey* method if the user chooses to make a selection by directly entering the customer or item ID.

Other class modules

The other classes are used to represent the business objects in this application like customers or items. The following section discusses the steps necessary to develop a Visual Basic program using the ActiveX bridge service. The topics that are covered include:

- Initializing the JVM
- Using a J2EE client container
- Accessing Java objects, methods, and fields
- Handling the errors

7.3 Initializing the JVM

The ActiveX client can create a JVM within its own process by calling the *XJBInit* method on a `JClassFactory` object. The *XJBInit* method takes an array of Strings as a parameter. This array of strings contain the same parameters that would be used when calling a Java program, listed between “Java” and the name of the class. Example 7-1 shows how to initialize a simple JVM in a Visual Basic application.

Example 7-1 Initialization of a simple JVM from a Visual Basic application

```
Dim oXJB as Object
Set oXJB = createObject("XJB.JClassFactory")
Dim astrJavaInitProps(0) as String
astrJavaInitProps(0) = "-Djava.class.path=.;c:\myjavaclasses"
oXJB.XJBInit(astrJavaInitProps)
```

In the Order Entry Client, the code is in the *XJBInit* method of the `ActiveX_XJBHelpers` module.

As a result of current restrictions on JNI, the JVM can’t be unloaded. It is there for the life of the ActiveX process. Since it can only be one JVM, there should only be one `oXJB` object. Storing it globally gives easy access to the JVM from any method. There is no harm in calling the code in Example 7-1 more than once, but it only serves to create a new `XJB.JClassFactory` with a connection to the same JVM resulting in needless overhead.

Now that the JVM has been initialized, the `oXJB` object can be used to access Java classes and objects.

7.4 Using a J2EE client container

Using a J2EE client container from within the attached JVM is optional, but it does provide many benefits and is the preferred approach.

7.4.1 Client container approach

The standard approach to developing a standard J2EE client side application is to run the Java application inside a J2EE client container. This provides:

- A programming model on the client side that is consistent with the server side.
- Enables easier lookup of EJBs and resources using the `java:comp` style naming. Example 7-2 shows how to lookup the home interface of the CustomerBean EJB when the client application runs inside a J2EE client container.

Example 7-2 Getting a reference to the CustomerHome

```
    Set oInitialContext = oXJB.NewInstance(oXJB.FindClass("javax.naming.InitialContext"))
    Set oJavaObj = oInitialContext.lookup("java:comp/env/ejb/CustomerBean")

    Dim oArgsb As Object
     Set oArgsb = oXJB.GetArgsContainer

     ' Narrow it
     ' - This is more EJB/J2EE code.  Here we use JMethodArgs object
     '   to coerce our EJB object to a specific class.   In this case
     '   our EJB Home class.
     ' - Notice that we are calling the static narrow() method on a Class Proxy
     '   object that was created from FindClass().
    Dim clsPortRemObj As Object
    Dim oCustomerHome As Object
    oArgsb.AddObject "java.lang.Object", oJavaObj
    oArgsb.AddObject "java.lang.Class",
oXJB.FindClass("com.ibm.itso.roch.wasaejb.CustomerHome")
    Set clsPortRemObj = oXJB.FindClass("javax.rmi.PortableRemoteObject")
    Set oCustomerHome = clsPortRemObj.narrow(oArgsb)
```

- Since the client JAR file can be packaged in the EAR file, along with the EJB archive and WAR files, everything can be distributed to the clients and servers in the same file.

The price for using this service is the time taken to initialize the container and launch the client (around 5 to 8 seconds).

Note: You can find more information on J2EE client containers in 4.7 “Java Clients” in the WebSphere Version 4 InfoCenter at:
http://www.ibm.com/software/webservers/appserv/doc/aee/index.html

7.4.2 No client container approach

The other approach is to run the client application inside a standard JVM but provide the JVM with the resource references needed to call the EJBs and server side services. The disadvantage of this approach is that the references have to be passed directly to the JVM when instantiating the `javax.naming.InitialContext` object and the application code will be more substantial. Example 7-3 shows the code for looking up the Customer Entity EJB home interface without a client container.

Example 7-3 Example code to obtain an EJB reference without a client container

```
    Dim p As Object
    Set p = oXJB.NewInstance(oXJB.FindClass("java.util.Properties"))

    ' Create an XJB helper object that will facilitate calling methods
    ' of objects that take parameters that are inherited.
    Dim oArgs As Object
    Set oArgs = oXJB.GetArgsContainer

    ' Build our properties using the normal naming string constants
    Dim Context_PROVIDER_URL As String
    Dim Context_INITIAL_CONTEXT_FACTORY As String
    Context_PROVIDER_URL = oXJB.FindClass("javax.naming.Context").PROVIDER_URL
    Context_INITIAL_CONTEXT_FACTORY =
oXJB.FindClass("javax.naming.Context").INITIAL_CONTEXT_FACTORY

    ' PROVIDER_URL is the location of the Bootstrap Host
    XJBPutProperty oXJB, p, Context_PROVIDER_URL, "iiop://127.0.0.1"
    XJBPutProperty oXJB, p, Context_INITIAL_CONTEXT_FACTORY,
"com.ibm.websphere.naming.WsnInitialContextFactory"

    ' Create the initial context
    Dim oInitialContext As Object
    oArgs.Clear
    oArgs.AddObject "java.util.Hashtable", p
    Set oInitialContext = oXJB.NewInstance(oXJB.FindClass("javax.naming.InitialContext"),
oArgs)

    Dim oJavaObj As Object
    ' "CustomerGlobal" is the JNDI reference name for the Customer EJB.
    Set oJavaObj = oInitialContext.lookup("CustomerGlobal")

    Dim oArgsb As Object
    Set oArgsb = oXJB.GetArgsContainer

    ' Narrow it
    ' - This is more EJB/J2EE code.  Here we use or JMethodArgs object
    '   to coerce our EJB object to a specific class.   In this case
    '   our EJB Home class.
    ' - Notice that we are calling the static narrow() method on a Class Proxy
    '   object that was created from FindClass().
    Dim clsPortRemObj As Object
    Dim oCustomerHome As Object
    oArgsb.AddObject "java.lang.Object", oJavaObj
    oArgsb.AddObject "java.lang.Class",
oXJB.FindClass("com.ibm.itso.roch.wasaejb.CustomerHome")
    Set clsPortRemObj = oXJB.FindClass("javax.rmi.PortableRemoteObject")
    Set oCustomerHome = clsPortRemObj.narrow(oArgsb)
```

This is not the suggested approach, and there is a good chance it will be deprecated in the next release. The performance is very similar regardless of wether the client container is used. The only difference is the time taken to launch and initialize the client container, which is only a few seconds.

7.4.3 Optimized container approach

An alternative is to take a slimmed down version of the full EAR file that was deployed on the server and use that as the reference when launching the client container. This approach would be used when the size of the whole EAR file would be an issue or when the whole EAR file contains secure information.

It is best to use the Application Assembly Tool (AAT) to remove unnecessary components such as WAR files. By doing this, the tool will also remove the components referenced in the deployment descriptors.

7.4.4 Components of the client side EAR file

In order for the ActiveX application to obtain the home interface of an EJB, it needs the compiled EJB stubs on the classpath. Generally the stubs are created during development and are, therefore, part of the EAR file. However if it is considered impractical or unsecure to hold the whole EAR file on each client workstation, then the stubs can be extracted from the EAR file and added to the classpath manually. If the whole EAR file is added to the client workstation, then the location of the file is passed to the client container when it is initialized within the ActiveX process. If only the stubs are installed on the client, then the ActiveX process doesn't need to initialize a client container within the JVM and can reference the EJB home interface via the full JNDI name.

7.4.5 Launching a standard Java J2EE application in a client container

To run a client application in a client container using the standard J2EE Application Client, the launchclient.bat file is run. It first calls the setupCmdLine.bat file to add various environment variables before running the client inside the client container using the JRE provided with the J2EE Application Client.

7.4.6 Launching a client container in an ActiveX process

To give the ActiveX client the access to the J2EE application client container, the launchClientXJB batch file is run:

```
launchClientXJB myClientApp.exe
```

Since this command needs to be run each time the application is restarted, it may be worth creating a simple batch file with contents similar to:

```
C:\IBM\WebSphere\AppClient\Enterprise\bin\launchclientxjb myClientApp.exe
```

The launchClientXJB.bat file, in turn, calls setupCmdLineXJB.bat to set up the environment variables. But unlike the standard **`launchClient`** command, it does not subsequently start the ActiveX application inside a client container. Instead it simply executes the application as is. The additional environment variables and addition of the J2EE Application Client JRE to the PATH enables the ActiveX application to subsequently initialize a JVM within its own process and subsequently launch a client container within that JVM.

The code that starts the J2EE client container is in the *XJBGetJ2EEClientContainer* method in the ActiveX_XJBHelpers module (Example 7-4). After the JVM is initialized, a `com.ibm.websphere.client.applicationclient.launchClient` object is instantiated, and the *launch* method is invoked passing in string parameters:

- Bootstrap host
- Bootstrap port

- Client JAR filename (if there is more than one in the EAR file; otherwise, it is set to an empty string)
- `initonly` (set to "true" to stop the container calling the Main class (see 7.8.1, "Setting up the Main class" on page 148))
- EAR file name
- EAR file extract directory
- Debug trace parameters

Example 7-4 The code in ActiveX_XJBHelpers that launches the client container

```
Public Function XJBGetJ2EEClientContainer(oXJB As Object, strEARFile As String, _
                            strBootstrapHost As String, strBootstrapPort As String, _
                            strAppClientJar As String, strEARTempDir As String)
    Dim oProps As Object
    Dim oCC As Object
    Dim astrAppParams(0) As String

    ' Initialize the J2EE Client Container

    ' Before initializing the Client Container, set the
    ' System Property that tells it what directory to
    ' extract the EAR file to.  If we don't do this, then
    ' a new directory will be created each time we launch the
    ' Client Container and will not get cleaned-up.
    If Trim(strEARTempDir) <> "" Then
        XJBPutProperty oXJB, oXJB.FindClass("java.lang.System").getProperties, _
                       "com.ibm.websphere.client.applicationclient.archivedir", _
                       strEARTempDir
    End If

    ' - Here we are creating a launchClient object which is supplied
    '   in appclient.jar supplied with WebSphere (Client or Server install).
    ' - Notice that we are eliminating the need to store the Class Proxy object
    '   returned by FindClass.  We are using as a temporary variable and immediately
    '   creating a new instance (Object Proxy) and throwing the Class Proxy away.
    Set oCC =
oXJB.NewInstance(oXJB.FindClass("com.ibm.websphere.client.applicationclient.launchClient"))

    ' Create a java.util.Properties object to use in our launchClient() J2EE API
    Set oProps = oXJB.NewInstance(oXJB.FindClass("java.util.Properties"))

    ' Add properties to the Properties object.
    ' - initonly=true is required to avoid running the Java Client that's stored
    '   in the EAR file.
    ' - BootstrapHost = "xxxx" : The name of the WebSphere server where your
    '                            EJB's are stored.
    ' - BootstrapPort = nnnn   : The port of the listener on the WebSphere server.
    ' - jar = "xxxx"           : The name of the jar within the ear file that
    '                            contains the Java Client (if you have more than one
    '                            Java Client Jar in the same EAR file.
    '                           (the path is relative to the root of the EAR file)
    XJBPutProperty oXJB, oProps, "initonly", "true"
    If Trim(strBootstrapHost) <> "" Then
        XJBPutProperty oXJB, oProps, "BootstrapHost", strBootstrapHost
    End If
    If IsNumeric(strBootstrapPort) Then
        XJBPutProperty oXJB, oProps, "BootstrapPort", strBootstrapPort
```

```
    End If
    If Trim(strAppClientJar) <> "" Then
        XJBPutProperty oXJB, oProps, "jar", strAppClientJar
    End If

    ' If we want to turn tracing on, we can do something like this:
    'XJBPutProperty oXJB, oProps, "trace",
"com.ibm.ws.client.applicationclient.ApplicationClientMetaData=debug=enabled"
    'XJBPutProperty oXJB, oProps, "tracefile", "c:\xjb.log"

    DoEvents

    ' Launch the client container.
    ' In the java world, it would start the Java Application within the
    ' container.  Here, however, we ARE the application, so we avoid
    ' starting it by setting the property initonly=true (above).
    ' The Client Container overrides the Java Namespace so that we can load
    ' the appropriate EJB class files from within our supplied ear file.
    oCC.launch strEARFile, oProps, astrAppParams

    Set XJBGetJ2EEClientContainer = oCC

    Set oCC = Nothing
    Set oProps = Nothing
    Exit Function
```

7.5 Accessing Java objects, methods, and fields

After an ActiveX client program has initialized the JVM, `XJB.JClassFactory` can be used to obtain access to Java objects and classes using `JObjectProxy` and `JClassProxy` objects. When accessing a Java class or object, the real Java object exists in the JVM, and the automation container contains the proxy for that Java object. The ActiveX program can use the proxy object to access the Java classes and objects, fields, and methods.

The `XJB.JClassFactory` is the object from which all Java classes and objects can be accessed and instantiated.

7.5.1 Primitive data type conversions

The ActiveX to EJB bridge performs conversions for COM primitive data types between native Automation types and their corresponding Java primitive data types and wrapper classes as shown in Table 7-1 and Table 7-2. All other types are handled automatically by the `JObjectProxy` objects.

The client program performs primitive data type conversion through the COM IDispatch interface (use of the IUnknown interface is not directly supported).

Table 7-1 Visual Basic to Java type conversion

Visual Basic type	Variant type	Java type	Notes
Byte	VT_I1	byte	Bye in Visual Basic is unsigned, but is signed in Java. See Byte Helper Function
Boolean	VT_BOOL	boolean	

Visual Basic type	Variant type	Java type	Notes
Integer	VT_I2	short	
Long	VT_I4	int	
Currency	VT_CY	long	See Currency Helper Function
Single	VT_R4	float	
Double	VT_R8	double	
String	VT_BSTR	java.lang.String	
String	VT_BSTR	char	
Date	VT_DATE	n/a	Date is not supported as a primitive

Table 7-2 Java to Visual Basic type conversion

Java type	Variant type	Visual Basic type	Notes
byte java.lang.Byte	VT_I1	Byte	Byte in Visual Basic is unsigned, but is signed in Java. See Byte Helper Function
boolean java.lang.Boolean	VT_BOOL	Boolean	
short java.lang.Short	VT_I2	Integer	
int java.lang.Integer	VT_I4	Long	
long java.lang.Long	VT_CY	Currency	
float java.lang.Float	VT_R4	Single	
double java.lang.Double	VT_R8	Double	
java.lang.String	VT_BSTR	String	
char java.lang.Character	VT_BSTR	String	
n/a	VT_DATE	Date	Not available. java.util.Date objects are represented as normal Object Proxy objects

For example, to correctly invoke a Java *setNumberOfLegs* method, which takes a Java int as a parameter, the Visual Basic code would have to pass a COM Long primitive as the argument:

```
anAnimal.setNumberOfLegs(CLng(4))
```

Conversion helper functions are needed where automatic conversions are not possible.

Byte helper function

Because the Java `Byte` data type is signed (-127 through 128) and the Visual Basic `Byte` data type is unsigned (0 through 255), you need to convert unsigned `Bytes` to a Visual Basic `Integers`, which look like the Java signed byte. See Example 7-5.

Example 7-5 The Byte helper function

```
Private Function GetIntFromJavaByte(Byte jByte) as Integer
     GetIntFromJavaByte = (CInt(jByte) + 128) Mod 256 - 128
End Function
```

Currency helper function

Visual Basic 6.0 cannot properly handle 64-bit integers like Java can (as the Long data type). Therefore, Visual Basic uses the `Currency` type, which is intrinsically a 64-bit data type. The only side effect of using the `Currency` type (the Variant type VT_CY) is that a decimal point is inserted into the type. Example 7-6 shows a method to extract and manipulate the 64-bit long value in Visual Basic.

Example 7-6 The Currency helper functions

```
' Currency Helper Types
Private Type MungeCurr
     Value As Currency
End Type
Private Type Munge2Long
     LoValue As Long
     HiValue As Long
End Type
' Currency Helper Functions
Private Function CurrToText(ByVal Value As Currency) As String
     Dim Temp As String, L As Long
     Temp = Format$(Value, "#.0000")
     L = Len(Temp)
     Temp = Left$(Temp, L - 5) & Right$(Temp, 4)
     Do While Len(Temp) > 1 And Left$(Temp, 1) = "0"
     Temp = Mid$(Temp, 2)
     Loop
     Do While Len(Temp) > 2 And Left$(Temp, 2) = "-0"
     Temp = "-" & Mid$(Temp, 3)
     Loop
     CurrToText = Temp
End Function
Private Function TextToCurr(ByVal Value As String) As Currency
     Dim L As Long, Negative As Boolean
     Value = Trim$(Value)
     If Left$(Value, 1) = "-" Then
         Negative = True
         Value = Mid$(Value, 2)
     End If
     L = Len(Value)
     If L < 4 Then
         TextToCurr = CCur(IIf(Negative, "-0.", "0.") & _
                 Right$("0000" & Value, 4))
     Else
         TextToCurr = CCur(IIf(Negative, "-", "") & _
                 Left$(Value, L - 4) & "." & Right$(Value, 4))
```

```
     End If
End Function
' Java Long as Currency Usage Example
Dim LC As MungeCurr
Dim L2 As Munge2Long
' Assign a Currency Value (really a Java Long)
' to the MungeCurr type variable
LC.Value = cyTestIn
' Coerce the value to the Munge2Long type variable
LSet L2 = LC
' Perform some operation on the value, now that we
' have it available in two 32-bit chunks
L2.LoValue = L2.LoValue + 1
' Coerce the Munge value back into a currency value
LSet LC = L2
cyTestIn = LC.Value
```

7.5.2 Accessing a Java class

The *FindClass* method is used to access a Java class. It takes the full name of the class as a parameter and returns a `JClassProxy` object:

```
Dim myClassProxy as Object
Set myClassProxy = oXJB.FindClass("com.myApp.MyClass")
```

`myClassProxy` can be used in the same way as the `MyClass` to access static methods and fields. Example 7-7 shows how to access class static methods.

Example 7-7 Invoking Java class methods through the JClassProxy

```
Dim myClassProxy As Object
Dim myString As String
Set myClassProxy = oXJB.FindClass("com.myApp.MyClass")
myString = myClassProxy.getName()
```

Example 7-8 shows how to access class static fields.

Example 7-8 Accessing Java class fields through the JClassProxy

```
Dim myClassProxy As Object
Dim myString As String
Set myClassProxy = oXJB.FindClass("com.myApp.MyClass")
myString = myClassProxy.name
```

In the above examples, the *getName* method and name attribute of `MyClass` return a `java.lang.String` object. This is automatically converted to a Visual Basic `String`.

7.5.3 Accessing Java objects

The *NewInstance* method of the `XJB.JClassFactory` object is called to create a new instance of a class. The method takes the `JClassProxy` object as a parameter and returns `JObjectProxy`, which is used to access the methods and fields of the new instance. The bridge is responsible for maintaining a hash table that associates the `JObjectProxy` object with the original Java object. Example 7-9 shows how to call the no parameter constructor of `MyClass`.

Example 7-9 Instantiating myObject via the no parameter constructor

```
Dim myObjectProxy As Object
Set myObejctProxy = oXJB.NewInstance(myClassProxy)
```

To call a constructor with parameters, the arguments are listed after `JClassProxy` in the call to *NewInstance* (see Example 7-10).

Example 7-10 Instantiating myObject via a constructor that takes java.lang.String as a parameter

```
Dim myObjectProxy As Object
Set myObjectProxy = oXJB.NewInstance(myClassProxy, "StringParameter")
```

This time the Visual Basic String object "StringParameter" is automatically converted to an instance of `java.lang.String`.

7.5.4 Invoking Java methods

There are several issues you need to deal with while calling Java methods:

- Case sensitivity
- Passing "Object" type parameters
- Processing the method's return value

Case sensitivity

Java methods are case sensitive, where ActiveX methods are not.

PowerBuilder doesn't support case sensitive method calls. All calls are converted to lowercase at runtime. This causes problems with the `JClassFactory` methods like *FindClass*.

Note: There is a workaround way to solve the problem with PowerBuilder. Refer to the following Web page:
http://www7b.boulder.ibm.com/wsdd/library/samples/powerbuilder.html

Visual Basic and VBScript also try to convert the case of some methods. However the case of the method is also changed dynamically in the VB developer tool, so the problem with method names can be identified before runtime. If you type the following line of the code, VB developer tool changes it:

```
myApp.close(123)
```

It is then changed to the following line when the cursor is moved to the next line:

```
myApp.Close(123)
```

The way to get around this problem in Visual Basic is to use the `CallByName` function:

```
CallByName(myApp, "close", vbMethod, 123)
```

In VBScript, a similar function called *Eval* does the same job:

```
Eval("myApp.close(123)")
```

Similarly Visual Basic may change the case of field references:

```
myApp.create = 123
```

This is converted to:

```
myApp.Create = 123
```

In Visual Basic, the `CallByName` function is used in the same way to reference the data fields that you would call a method:

```
CallByName(myApp, "create", vbLet, 123)
```

Here *vbLet* is used for primitive COM types and *vbSet* for objects.

In VBScript, use:

```
Eval("myApp.create = 123")
```

The `CallByName` function doesn't handle Java exceptions in the standard way (see 7.6, "Error handling" on page 142). Instead it throws errors with Err.Number of 438, and the Err.Description field that doesn't contain the Java exception stack trace. This means that any valid exceptions have to be caught and handled in the Java code, or an error code has to be returned as the return value from the method.

Passing 'Object' type parameters

The ActiveX bridge uses the object type of the arguments passed to a method to find the method on the Java object or class that is intended to be invoked. Because of polymorphism, sometimes the type of the parameter passed doesn't match the parameter in the method signature. For example, a method specifies its parameter type as `java.lang.Object`. If you try to invoke such a method, passing a parameter of other type (which inherits from `java.lang.Object`), a COM error with the description "object does not support this property or method" is thrown.

To cater to this, the ActiveX bridge API provides a container for such parameters. The container, `XJB.JMethodArgs`, is retrieved using the *GetArgsContainer* method on the `JClassFactory` object. Each parameter can be added to the `JMethodArgs` object by using the *AddObject* method. The first parameter of this method is the type declared in the target method's signature, and the second parameter is the object itself. *AddObject* can be called multiple times until it contains all the parameters destined for the target method. The order in which the objects are added to `JMethodArgs` must be the same as the order of parameters in the target method signature. The *Clear* method in the `JMethodArgs` object removes all objects, preparing it for re-use. Example 7-11 shows the situation where the *equals* method of a `java.lang.String` object is called. This method has the signature:

```
public boolean equals(Object anObject)
```

Therefore passing `java.lang.String` as a parameter would cause an error. The `stringb` argument must be contained in a `JMethodArgs` object and labeled "java.lang.Object".

Example 7-11 Passing a parameter to a method in an JMethodArgs object

```
Dim StringClassProxy As Object
Dim stringa As Object
Dim stringb As Object

Set StringClassProxy = oXJB.FindClass("java.lang.String")

Set stringa = oXJB.NewInstance(StringClassProxy, "Bill")
Set stringb = oXJB.NewInstance(StringClassProxy, "Ted")

Dim args As Object
Set args = oXJB.GetArgsContainer
```

```
args.AddObject ("java.lang.Object", stringb)

If (stringa.equals(args)) Then
...
End If

‘Reuse the same argument container object

args.Clear
...
```

Return values

Although the signature of a Java method declares the return type, the method may return a child or an implementor of that type. The bridge uses the Reflection API to examine or “introspect” the return value to determine its actual data type. If the actual data type corresponds to a COM primitive (see Table 7-1 and Table 7-2), the bridge will convert the object to the associated COM primitive and pass the primitive to the client application. The return value should then be assigned to a variable of the appropriate type.

If the return type does not correspond to a COM primitive, either a `JObjectProxy` or `JClassProxy` object is returned to the ActiveX process. Both `JObjectProxy` or `JClassProxy` objects can be assigned to a variable declared as type `Object` in the Visual Basic code.

Example 7-12 demonstrates handling both primitive and Object return types:

1. The penultimate line of code calls the *getTime* method on a `JObjectProxy` object representing a `java.util.Calendar` instance. The *getTime* method of the `java.util.Calendar` instance returns a `java.util.Date` object, although the bridge returns a `JObjectProxy` object representing the `Date` object. The code then assigns that `JObjectProxy` object to `oDate` that was declared as type `Object`.
2. The *getTime* method of `oDate` is then called and the bridge invokes the *getTime* method of the `java.util.Date` object, which returns a Java primitive `int`. The bridge converts this `int` to the corresponding COM primitive, `Long`. The code then assigns this value to the variable `time`, which was declared as type `Long`.

Example 7-12 Handling return values

```
Dim CalendarClassProxy As Object
Dim oCalendar As Object
Dim oDate As Object
Dim time As Long

Set CalendarClassProxy = oXJB.FindClass("java.util.Calendar")

Set oCalendar = CalendarClassProxy.getInstance

Set oDate = oCalendar.getTime

time = oDate.getTime
```

7.5.5 Handling arrays

Handling arrays in automation containers is similar to handling arrays in Java.

When Java arrays are passed to ActiveX, their elements are converted to `JObjectProxy` objects or according to the Data Conversion Table if they correspond to COM primitives. The array is then wrapped in a variant array before being passed to the automation container. For this reason, the result of a call to a Java object that returns an array must be assigned to a variant or a variant array. The use of variants with arrays is needed so the same bridge can be used to support both Visual Basic and VBScript.

The retuned variant can't be interrogated to discover the data type. However, since the bridge can determine the actual data type enclosed in the variant wrapper, the array can be passed as a parameter to other Java method calls. Methods calls on the variant element will also invoke the appropriate method on the Java object.

Example 7-13 shows the *GetAllCustomer* method in the `COrderClerk` module of the Visual Basic Order Entry Client. `Clerk` is the variable that references the `OrderEnrtyClerk` session EJB. `Clerk.findAllCustomers.elements()` calls the *findAllCustomers* method on the EJB, and then the *elements* method is called to return an enumeration of the results. A `JObjectProxy` object, which represents `enumeration`, is passed back to the Visual Basic client application. The `enumeration` contains a set of `String` arrays. We access each `String` array by calling the *nextElement* method in `enumeration`. However the signature of the *nextElement* method declares that the return value is of type `Object`. The ActiveX bridge realizes that the object returned by `CustomerList.nextElement` is actually a `String` array and handles this by returning a `Variant` array to the application. Each element of the `String` array can be assigned to a `String` in the code.

The error handling around the call to `Clerk.findAllCustomers.elements()` caters to the session EJB timing out.

Example 7-13 Handling arrays associated with COM primitives

```
Public Function GetAllCustomers() As Variant

    '-------------------------------------------------------
    'Get all customers, return as an enumeration of strings
    '-------------------------------------------------------

    Dim CustomerList As Object

    '-------------------------------------------------------
    'Get all customers, put into enumeration object
    '-------------------------------------------------------
    On Error Resume Next

    Set CustomerList = Clerk.findAllCustomers.elements()

    If ((Err.Number = &H6003) And _
        (InStr(Err.Description, "java.rmi.NoSuchObjectException") > 0)) Then
          Dim oCHome As Object
            Set oCHome = g_Config.EJBHomeRefs.OrderEntryClerkHome
            Set Clerk = CallByName(oCHome, "create", VbMethod)
            Set CustomerList = Clerk.findAllCustomers.elements()
        ElseIf (Err.Number <> 0) Then
        GoTo ErrorHandler
    End If
    '-------------------------------------------------------------------------------
```

```
    'iterate through recordset, add to customer collection
    '-------------------------------------------------------------------------------
    Call Customers.Clear

     Dim TempCustomer() As Variant

     Do While CustomerList.hasMoreElements()

      TempCustomer = CustomerList.nextElement

      Set CustomerObject = Customers.Add(Trim(TempCustomer(CustomerData.custID)), _
                                  Trim(TempCustomer(CustomerData.FirstName)), _
                                  Trim(TempCustomer(CustomerData.LastName)), _
                                  Trim(TempCustomer(CustomerData.MiddleInitials)), _
                                  Trim(TempCustomer(CustomerData.Address1)), _
                                  Trim(TempCustomer(CustomerData.Address2)), _
                                  Trim(TempCustomer(CustomerData.City)), _
                                  Trim(TempCustomer(CustomerData.State)), _
                                  Trim(TempCustomer(CustomerData.ZipCode)))

    Loop

    Set GetAllCustomers = Customers

    Exit Function

ErrorHandler:

     Call HandleErrors(ROUTINE_NAME)

End Function
```

There are some other aspects of using arrays:

- In Java, all the elements of an array must be of the same type. For this reason, a `Variant` array must contain all elements of the same type if it is to be included as a parameter in a Java method call.
- Java arrays are always zero-based, meaning the first element is always at position 0 when it is received in an automation container.
- Nested or multi-dimensional arrays are treated as zero-based multi-dimensional arrays in Visual Basic or VBScript.

7.5.6 Threads

The ActiveX to EJB bridge supports both free-threaded and apartment-threaded access and implements the Free Threaded Marshaler to work in a hybrid environment such as Active Server Pages. Each thread that is created in the ActiveX process will be mirrored in the Java environment when the thread communicates through the ActiveX to EJB bridge.

In addition, once all references to Java objects (there are no `JObjectProxy` or `JClassProxy` objects) in an ActiveX thread are released, the ActiveX to EJB bridge detaches the corresponding thread from the JVM. Therefore, you must be careful that any Java code that you access from a multi-threaded Windows application is thread-safe. Visual Basic and

VBScript applications are essentially single-threaded. Therefore, Visual Basic and VBScript applications don't really need to worry about threading issues in the Java programs they access. With Active Server Pages, you need to make sure that your Java program is thread-safe.

7.6 Error handling

All exceptions thrown by Java are encapsulated and re-thrown as COM errors and appear as `Err` objects in Visual Basic and VBScript. Java exceptions are more complex than Visual Basic `Err` objects, which only contain error number and description fields. The details of the Java exception are written to the two Err fields in Table 7-3.

Table 7-3 Correlation between COM errors and Java exceptions

Error number (Hex)	Error number (decimal representation)	Description
0x6001	24595	JNI error
0x6002	24596	Initialization error
0x6003	24597	Java exception Error description is the Java Stack Trace.
0x6FFF	28671	General Internal Failure

In Visual Basic and VBScript, the COM error numbers are converted to 2 byte longs stored in the Err.Number field. The values of these longs are the decimal representation of originating COM Hex value. For example, for Java exceptions, 0x6003 becomes 6 x16 x16 x16 + 3 = 24597. Other languages may use more memory to store this 2 byte Hex value changing its decimal representation.

In the OrderEntry application, errors are handled in two ways. The `If` statement determines if an error message should be displayed to the user. In the example shown in Example 7-14, an error message needs to be displayed if the Customer ID doesn't exist in the database. If this event occurs, a `javax.ejb.FinderException` exception is thrown with reference to the `Customer` EJB. `ErrorHandler` first uses the value 24597 to identify a Java exception and then the key words "FinderException" and "CustomerBean" to identify that the exception was caused by an invalid Customer ID. The code could also compare `Err.Number` to `&H6003`.

Example 7-14 GetCustomer function in COrderClerk.cls

```
Public Function GetCustomer(ByVal CustomerID) _
        As Object
....

    On Error GoTo ErrorHandler
....

ErrorHandler:
    '-------------------------------------------------------------------------------------
    'Identify a Customer not found error
    '-------------------------------------------------------------------------------------
    If (Err.Number = 24597) Then
        If ((InStr(Err.Description, "CustomerBean") > 0) And _
            (InStr(Err.Description, "javax.ejb.FinderException") > 0)) Then
          RaiseEvent CustomerError(Errors.ERRID_NO_RECORD_FOUND)
       End If
```

```
    End If
    '-------------------------------------------------------------------------------------
    'Log the error
    '-------------------------------------------------------------------------------------
    Call HandleErrors(ROUTINE_NAME)
End Function
```

HandleErrors(ROUTINE_NAME) is then called. This method uses the error description (Err.Description) to categorize and log the error (see Example 7-15).

Example 7-15 HandleErrors function in COrderClerk.cls

```
Private Sub HandleErrors(ByVal RoutineName)

    Dim ejbName
    Dim ejbNamePos
    Dim errDescription

    Const ROUTINE_NAME = "HandleErrors"

    '-----------------------------------------------------
    'handle message returned from JVM
    '-----------------------------------------------------
    Select Case True

        '-------------------------------------------------------------------------
        'EJB not deployed error
        '-------------------------------------------------------------------------
        Case InStr(Err.Description, "javax.naming.NameNotFoundException") > 0
            ejbNamePos = InStr(Err.Description, "Remaining Name:")
            ejbNamePos = ejbNamePos + 15
            ejbName = Mid(Err.Description, ejbNamePos)

            errDescription = "The Enterprise Java Bean " & _
                             ejbName & _
                             " is not deployed."

        '-------------------------------------------------------------------------
        'record not found (tried to get by key)
        '-------------------------------------------------------------------------
        Case InStr(Err.Description, "javax.ejb.FinderException") > 0
            Err.Number = Errors.ERRID_NO_RECORD_FOUND

        Case InStr(Err.Description, "javax.ejb.ObjectNotFoundException") > 0
            Err.Number = Errors.ERRID_NO_RECORD_FOUND

        '-------------------------------------------------------------------------
        'other error
        '-------------------------------------------------------------------------
        Case Else
            errDescription = Err.Description

    End Select

    '-----------------------------------------------------
    'add message to Errors collection to return to caller
    '-----------------------------------------------------
    Errors.AddError ModuleName:=MODULE_NAME, _
```

```
                    RoutineName:=RoutineName, _
                    ErrorNumber:=Err.Number, _
                    ErrorDescription:=errDescription, _
                    ErrorSource:=Err.Source

End Sub
```

7.7 Good programming guidelines

Keep the interface simple. The quickest way to access Java components and resources is by using the Java language. Therefore, we recommend that the majority of code is written in Java, and the application interface is limited to the fundamental methods (see the discussion in 7.2.1, “Designing an ActiveX client program” on page 124).

7.7.1 Visual Basic guidelines

There are several generic guidelines for writing and deploying an ActiveX client program.

Launching the application or IDE

In order to give your IDE access to the ActiveX bridge, you must launch the IDE after setting the environment variables set by running the setupCmdLineXJB batch file. Failure to do this prevents the ActiveX process from being able to initialize a JVM. The easiest way to do this is to precede the IDE executable or the name of the ActiveX application with the launchClientXJB batch file, which calls setupCmdLineXJB before launching the IDE. The batch files are in the WebSphere install directory <WebSphere_install_directory>\WebSphere\AppClient\Enterprise\bin\.

For example, if the IDE is Visual Basic Version 6, use a DOS prompt to navigate to the location of the batch files. Then, enter either of the following commands:

```
launchClientXJB VB6.EXE
launchClientXJB MyApplication.vbp
```

It is very similar for VBScript files. You can enter either of the following lines:

```
launchClientXJB MyScript.vbs
launchClientXJB script Mysrcipt.vbs
```

Storing the XJB.ClassFactory globally

Once a JVM is loaded into an ActiveX process, it can’t be unloaded and re-initialized. Therefore, it makes sense to keep the reference to the `XJB.ClassFactory` throughout the lifetime of the application either as a global variable or by passing the reference as a method parameter. There is also a performance impact of calling `createObject(“XJB.ClassFactory”)` and re-running the code, more than once, to create and initialize the `XJB.ClassFactory` object.

In the VB OrderEntry Client Application, the `XJB.ClassFactory` object is stored in the `g_Config` object, which is stored globally. All references are made using `g_Config.XJB`. By doing this, `g_Config` can ensure the JVM has been initialized before it is returned.

Re-initializing the JVM

If changes are made to the JVM initialization parameters, Visual Basic or the ActiveX process has to be restarted in order to bring those changes into effect. If Visual Basic is not restarted, the `XJBInit` function just returns the original JVM.

7.7.2 Active Server Pages guidelines

To optimize the usage of the ActiveX bridge with the ActiveX programs, use the guidelines in the following sections.

Unloading the ASP application

If the JVM needs to be re-initialized, due to changes in the client side EAR file or JVM initialization parameters, the ASP application has to be restarted. This is done by using the Unload button in the Internet Information Server administration console in the ASP application properties (Figure 7-5 and Figure 7-6).

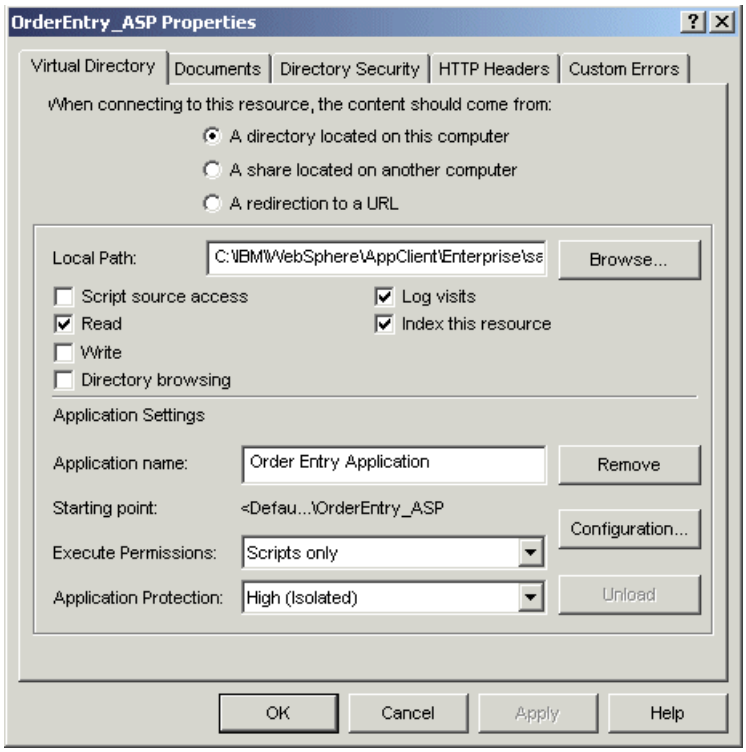


Figure 7-5 Windows 2000: Properties window for an ASP application

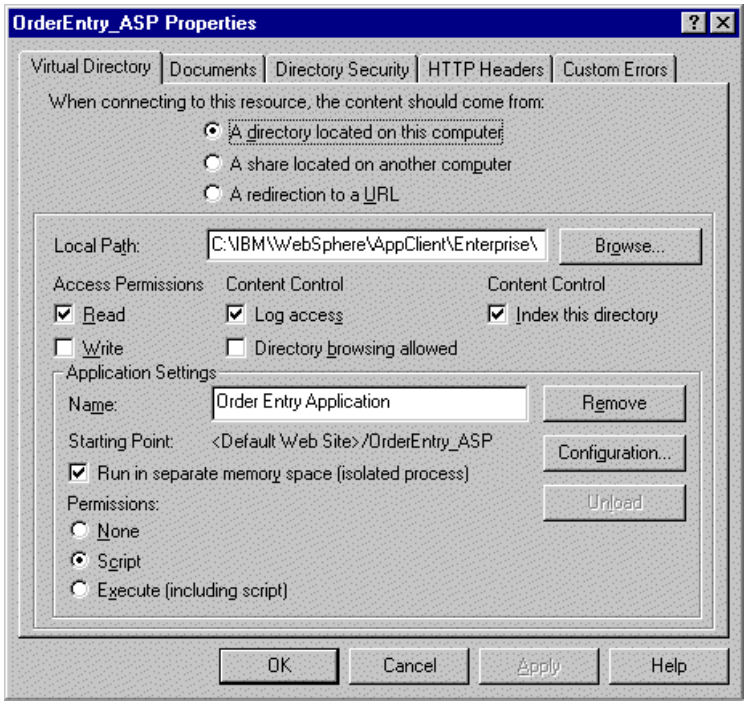


Figure 7-6 Windows NT: Properties window for an ASP application

Isolating each ASP application

If there is more than one JVM to which different ASP applications have to connect, each application has to run in its own process. The way to achieve this is to set the Application Protection variable in the ASP Application properties to High (Isolated) in Windows 2000 as shown in Figure 7-5, or set as an isolated process in Windows NT (Figure 7-6).

Adding the WebSphere Client JRE to the system PATH

At the point when the ASP application calls *XJBInit* on the `XJB.JClassFactory` object, the JRE runtime directories have to be on the system path so the Java DLLs can be found. Since ASP applications don't have an individual PATH, the following addition must be made to the system PATH before the application is run:

```
%JAVA_HOME%\bin;%JAVA_HOME%\bin\classic
```

Here *%JAVA_HOME%* is where the home of the WebSphere Client JRE (for example, C:\WebSphere\AppClient\java\jre).

This addition to the system PATH is made in the SetUpCmdLineXJB batch file, but this is not run for ASP applications. A result of adding the JRE runtime to the system PATH is that the same JVM version will be used for all the ASP applications. You must reboot the system, running the Internet Information Server, after you change the system's PATH.

Storing the XJB.JClassFactory in the Application Scope

As a result of the one-to-one relationship between the JVM and the ActiveX process, the fact that the JVM remains attached to the process throughout its life cycle means there is no need to call *XJBInit()* more than once. The resulting `XJB.JClassFactory` object should be stored at the application scope. Example 7-16 shows part of the *SetupApplicationConfig* method of an ASP application that uses `Application("oXJB")` and `Application("oCC")` to store the XJB and Client Container objects in application scope. Note that an application lock is used to ensure that both objects are in the same thread.

Example 7-16 Code showing oXJB and oCC being stored in application scope

```
Private Sub SetupApplicationConfig(strUserid, strPassword)
   ' We are globalizing XJB and the Client Container object to speed things up.
   ' Since you can only have one JVM and J2EE Client Container per process
   ' and also because we can't destroy the JVM, it makes sense to keep these
   ' global and continue to reuse them.

   ' Initialize the XJB Environment (the JVM) and Client Container
   ' Only Once.
   ' Use an Application lock to ensure that both objects are
   ' in the same thread.
   Application.Lock
   If IsEmpty(Application("oXJB")) Then
      AddMessage("Initializing XJB.JClassFactory")
       Set Application("oXJB") = XJBInit(strWAS_HOME, strJAVA_HOME, strNAMING_FACTORY)
   End If

   If IsEmpty(Application("oCC")) Then
       Set Application("oCC") = XJBGetJ2EEClientContainer(Application("oXJB"), strEar,
strBootstrapHost, strBootstrapPort, "", strEarTempDir)

...
```

7.7.3 Client container guidelines

Follow these guidelines to optimize the usage of the J2EE client container.

Storing the client container as a global variable

As for the JVM, there can only be one client container per JVM. Best practice is to hold the client container as a global variable.

Re-initializing the client container

By default, when the client container is first initialized, by calling the container's *launch()* method, it extracts the EAR file into a random directory name in the system temporary directory. In a standard J2EE Java client, these files are then cleaned up when the container shuts down. When the ActiveX client is restarted and the client container is re-initialized, another random directory name is used to extract the EAR contents.

The way to get around this problem is to specify the `com.ibm.websphere.client.applicationclient.archivedir` system property in the JVM before the client container is initialized. Now if the EAR file is already extracted to that location, the container will re-use those files.

Important: If the client side EAR is changed, this directory must be deleted and the application restarted prior to the call to the container's *launch* method. Otherwise, the container will re-use the contents of the old EAR file.

As a result of the properties mentioned above, it is important to use a different archive directory for every EAR file referenced on the client workstation.

7.8 Modifying the EAR file for the ActiveX J2EE client container

In the standard J2EE client application model, the client code is contained in the Enterprise Application File (EAR file) as a JAR file. The same EAR file can then be deployed on the server and the client. Although different components are used by the client and the server, it makes sense to package everything together.

When the client application is launched in a client container, the application is given access to the resources required by the application including the EJB stubs.

In the case where the client application is an ActiveX component, there is no Java code needed for the client application. However, in order for the client container to be launched inside the JVM, the client container needs to know the name and location of the client application JAR file. For this reason, a "dummy" client JAR needs to be created.

7.8.1 Setting up the Main class

The client container not only needs to know the name and location of the "dummy" client JAR, it also mandates that the client JAR's configuration includes the name of the first class to be called when the client container is launched. This class is referred to as the *Main class* (it contains the *main* method, which is invoked by the client container). Since the client JAR contains no functionality, the `initonly` parameter of the client container is set to "true" by the ActiveX process before the container is launched. This prevents the Main class's *main* method from being invoked. However, even though this class is never called, the container will throw an error on launch unless the Main class exists.

You must follow these rules when setting up the Main class:

- The class must be in the client JAR.
- There must be an entry in the client's Manifest.MF to indicate the name and path of the file. For example, if the name and path is \MyApp\DummyClient.class with respect to the client JAR's root, the entry in the Manifest.MF file would be:

  ```
  Main-Class: MyApp.DummyClient
  ```

- The file must have the same name as the class it contains.
- It is not necessary for the class to have a *main* method, although the code for `DummyClient` in Example 7-17 does.

Example 7-17 The DummyClient class with the no op main()

```
public class DummyClient {
    public static void main (String[] args) {
    //Do nothing
    }
}
```

7.8.2 Creating the ‘dummy’ client JAR

The creation of the client JAR is the easiest to achieve using the Application Assembly Tool (AAT):

1. Start the AAT, and open the server EAR file.
2. Click the **Wizards** icon and select **Create Application Client** wizard.
3. Use the drop-down list next to **Containing Application** to select the EAR file.
4. Type a file name and display name.
5. Click **Next** (Figure 7-7).

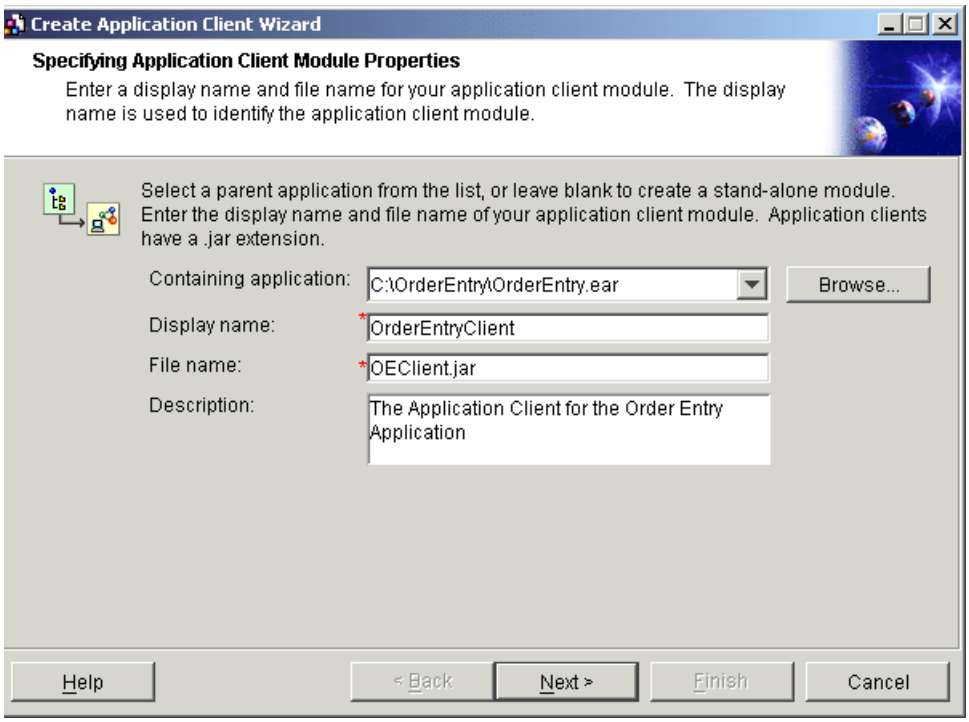

Create Application Client Wizard
Specifying Application Client Module Properties
Enter a display name and file name for your application client module. The display name is used to identify the application client module.
Select a parent application from the list, or leave blank to create a stand-alone module. Enter the display name and file name of your application client module. Application clients have a .jar extension.
Containing application:
C:\OrderEntry\OrderEntry.ear
Browse...
Display name:
OrderEntryClient
File name:
OEClient.jar
Description:
The Application Client for the Order Entry Application
Help
< Back
Next >
Finish
Cancel

Figure 7-7 Create Application Client Wizard in the Application Assembly Tool (AAT)

6. Add the `DummyClient` class (see 7.8.1, “Setting up the Main class” on page 148).
7. Click **Add**.
8. Click **Browse** and navigate to the directory where the `DummyClient` class is located (Figure 7-8).

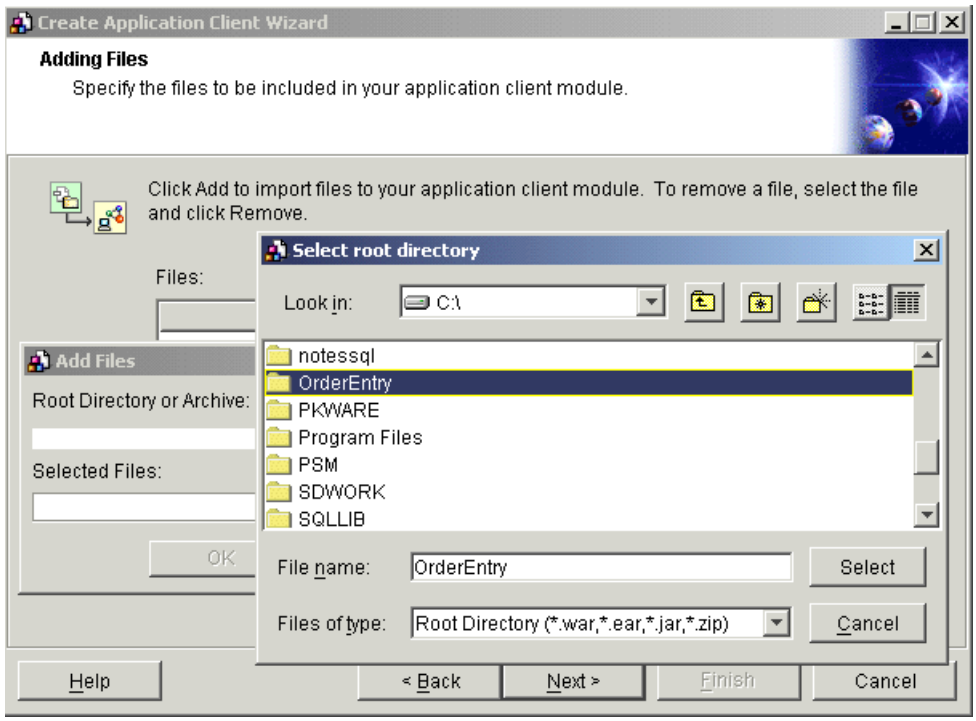


Figure 7-8 Adding the DummyClient.class as the Main class

9. Click **Select** when the directory is highlighted.
10. In the Add Files window, highlight the **DummyClient.class** and click **Add**.
11. Click **OK**.
12. Select or enter **DummyClient** as the Main Class in the next window.
13. Complete the rest of the wizard leaving the EJB References blank for the moment.

7.8.3 Adding the EJB references

It is easier to add the EJB references after the client Module has been completed because the tool now dynamically picks up the EJBs in the appropriate EJB module:

1. Right-click **EJB References** under the newly created **Dummy Client**.
2. Select **New**.
3. To add a reference, use the **Link** drop-down box to select the appropriate EJB. The AAT now dynamically updates the Home, Remote, and Type fields as shown in Figure 7-9.

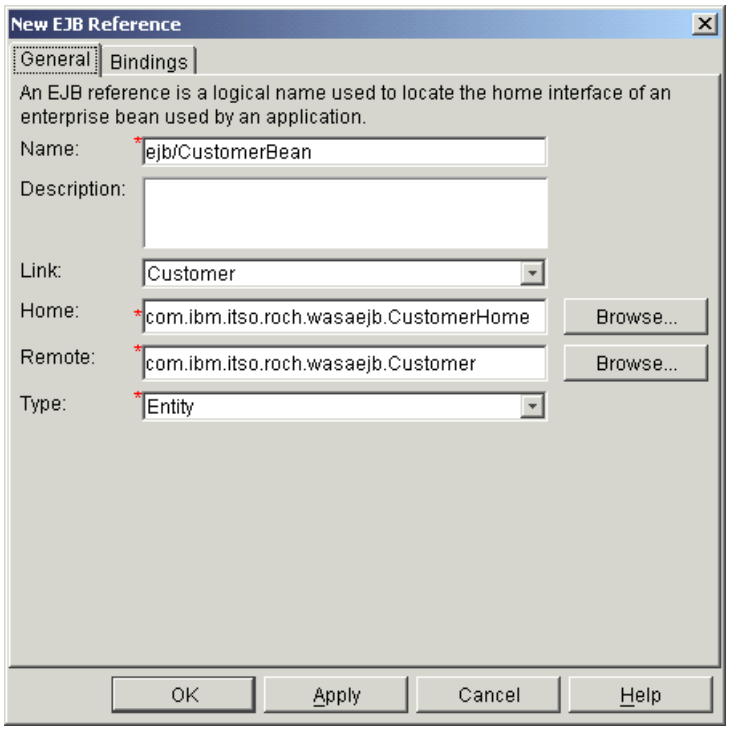


Figure 7-9 Adding EJB references to the client

4. Enter the name of the reference. The EJB is found from the ActiveX process using the reference “Name” preceded by `java:comp/env/`. For example, using the helper method *XJBGetEJBHome* to retrieve the Customer EJB’s home interface object in the OrderEntry application the code in the `ActiveX_EJBHomeRef` module (see Example 7-18).

Example 7-18 Getting a home interface object

```
Set CustomerHome = XJBGetEJBHome(oXJB, LocalContext, "java:comp/env/ejb/CustomerBean",
"com.ibm.itso.roch.wasaejb.CustomerHome")
```

5. Click the **Bindings** tab.
6. Type the JNDI name (the JNDI name of the referenced EJB). The JNDI name can be found by highlighting the corresponding EJB under the appropriate EJBModule in the left-hand pane. Then selecting the **Bindings** tab in the lower right-hand pane. The JNDI name is the actual location of the EJB in the JNDI Name Space.
7. Repeat this process for any other EJB reference required by the client application.

7.8.4 Optimizing the EAR file

If there is a concern with security or the size of EAR file, you can optimize the EAR file. The Application Assembly Tool can now be used to slim down the EAR file for use on the client. AAT should be used because it will also remove all references to the redundant components. The following components can be removed from the EAR file:

- Any WAR files
- Any DLLs or XML files related to building the application database
- Java source code: There may also be a requirement to remove this for security reasons
- Unreferenced EJB stubs and ties

8

The CORBA services

This chapter discusses:

- Positioning of CORBA
  - CORBA in general
  - Role of CORBA in J2EE
- Overview of WebSphere Application Server CORBA support
  - WebSphere support scenarios
  - Interoperability with EJB
  - C++ CORBA Software Development Kit (SDK)
- C++ CORBA clients for the OrderEntry application
  - Configuring the environment
  - IBM client of the Customer EJB
  - VisiBroker client of the Customer EJB
  - Problem determination
- Connecting the OrderEntry application to a C++ CORBA server
  - Configuring the environment
  - Connecting to an Orbix server
  - Problem determination

Note: To understand the structure of the OrderEntry application and to run it, read the instructions in Appendix A, “Downloading and installing the OrderEntry application” on page 221.

© Copyright IBM Corp. 2002

8.1 Positioning of CORBA

CORBA is the acronym for Common Object Request Broker Architecture as specified by the Object Management Group. It was adopted to provide a language- and vendor-neutral standard for object distribution. It hides location and network complexities from the programmer and provides services such as naming, transaction, and security.

The CORBA architecture is built around a special layer, the Object Request Broker (ORB), that facilitates communication between clients and server objects. A client ORB marshals request with parameters and demarshals the reply with results. A server ORB is responsible for handling the object request from a client by demarshalling a request, invoking the appropriate method on the service object, and marshalling the reply/results. Low-level communication between different object spaces (ORBs) is done by using the Internet-Inter ORB Protocol (IIOP).

When a client invokes a member function on a CORBA object, the ORB intercepts the function call. The ORB redirects the function call across the network to the target object. After the target object has created the response, the ORB then collects the result from the function call and return this to the client.

Figure 8-1 shows the role of CORBA for distributed communication.

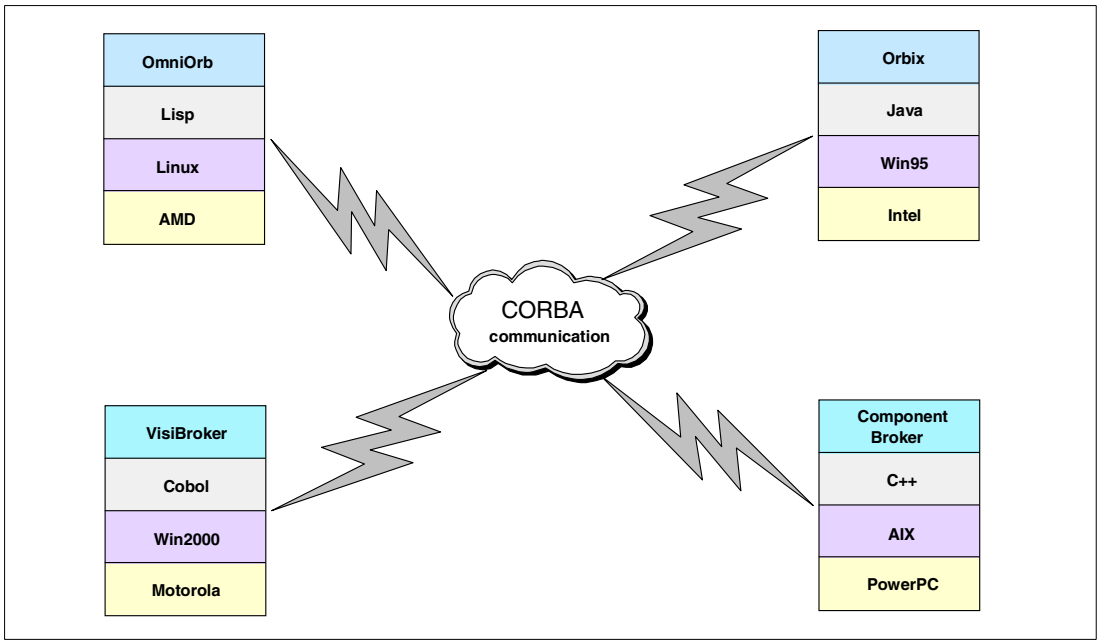


Figure 8-1 Role of CORBA communication

CORBA has been implemented by several vendors. The most wide-spread implementations are:

- Orbix (from IONA)
- VisiBroker (from Borland)
- ComponentBroker (from IBM)
- OmniOrb (from AT&T, available as open source)

Note: In practice, the interoperability between ORBs of different vendors has been poor for some time because the first releases of CORBA did not fully specify the server objects. The Basic Object Adapter (BOA) left a lot of room for interpretation. Since the Portable Object Adapter (POA) has been fully specified, interpretation has been reduced, but interoperability is still an issue.

Apart from the ORB, there are two other key building blocks in the CORBA model:

- Interface Definition Language (IDL)
- CORBA services

The IDL describes the object services and normalizes the differences caused by language or operating system dependencies. CORBA services provide standard ways for CORBA objects to interact. Examples of CORBA services include naming, transaction, event, and notification.

CORBA is independent of the programming language. To use the ORB, it is necessary for programmers to know how to access ORB functionality from their programming languages. CORBA Language Mapping specifications define the mapping of IDL constructs to several programming languages. There exist language mappings for Java, C++, C, Smalltalk, Ada, Lisp, COBOL, Python, and CORBA Scripting Language.

The most recent release of the specification is CORBA 2.5, but release 3.0 is planned to be finished in soon. But most of the vendors have products that do not comply to a newer release than 2.3 at the moment.

Locating a server object

A key topic in the CORBA model is how a client locates its server object. There are several ways to do this, but the standard way would be to use the CORBA Naming Service. The Naming Service allows you to bind one or more logical names to an object reference in a naming space. It also allows your client applications to use the Naming Service to obtain an object reference by using the logical name assigned to that object.

Figure 8-2 looks at the technical details involved in the process of binding a client to a server object (called *servant*).

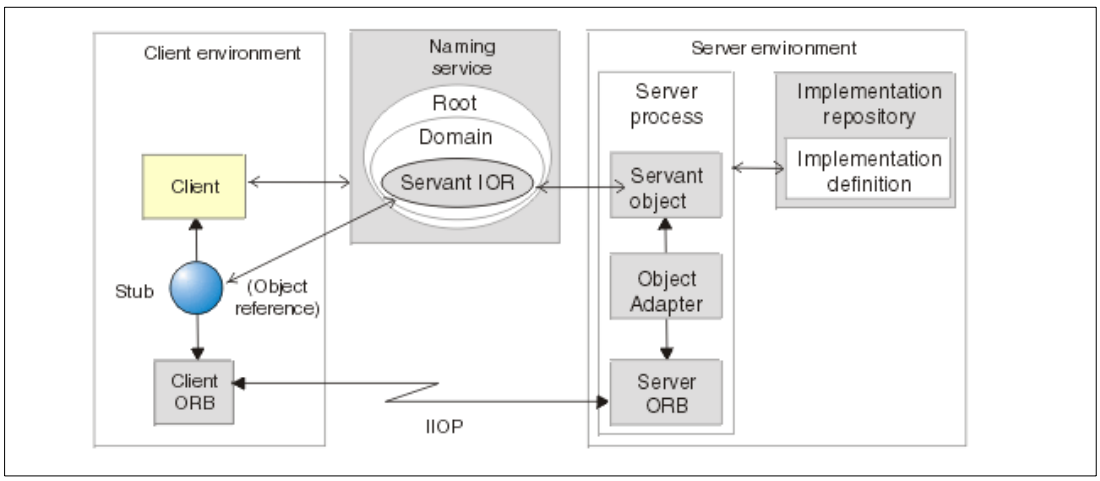


Figure 8-2 CORBA method call in detail

In the server environment, there are basically the server process and the implementation repository, which stores information about all server processes. The server process itself exists of the server ORB, the Object Adapter, and the servant objects. The Object Adapter (BOA or POA) dispatches the incoming requests from the ORB to the various implementation objects (servants).

When the server process is started, it has to export the references of the objects it hosts. One way is to register the objects with a naming service. The naming service stores an object reference in the form of an *interoperable object reference (IOR)*. An IOR is a string representation of an object reference that can be exchanged between two ORBs. Another way to export object references is to write them to files.

The client environment contains the client application code, the stub, and the client ORB. The stub is a proxy object for the servant, which behaves to the client code like a local object. It does the marshalling (translates method call for network transport) and directs the requests to the ORB.

When a client wants to call a method on a servant object, it first locates an object reference via the naming service. With this object reference, the stub object is instantiated. Then the client application can invoke the method in the servant.

CORBA development

Figure 8-3 shows an overview about the development steps for CORBA.

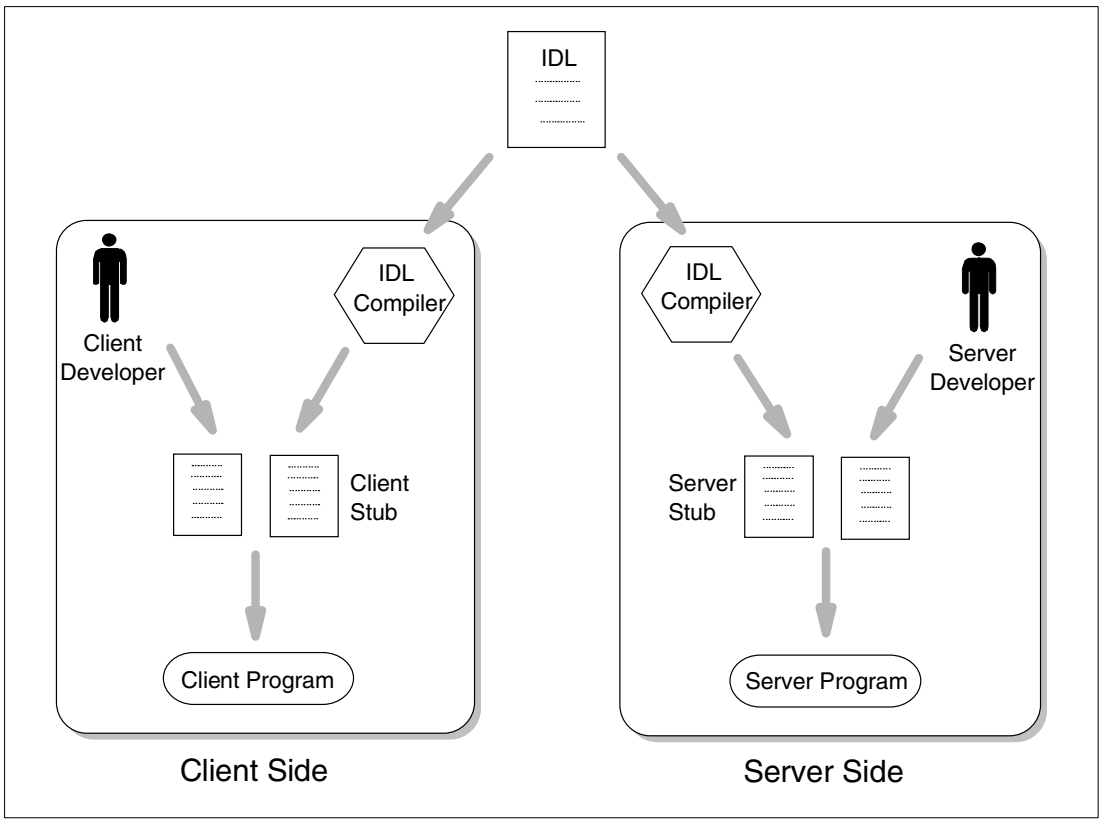


Figure 8-3 CORBA development steps

To develop a CORBA application, you have to perform the following steps:

1. Define the IDL interfaces. Identify the objects required by the application, and define their public interfaces in IDL.

2. Compile the IDL files with an IDL compiler. This creates the client and server stub files. Depending on the IDL compiler, you may already get a dummy implementation for the client and the server.
3. Develop the server program. The server acts as a container for a variety of CORBA objects, each of which supports one IDL interface. The server developer must add code to provide the business logic for each type of CORBA object. The server makes its CORBA objects available to clients by exporting object references to a well-known location.
4. Develop the client program. The client uses the IDL compiler-generated mappings to invoke operations on the object references that it obtains from the server.
5. Build the application with a language specific compiler, linker, and archiver.

You can find further reading material at: http://www.omg.org

8.1.1 Role of CORBA in J2EE

This section discusses what CORBA is used for and how to use it within the J2EE platform.

The task of CORBA

CORBA is the J2EE distributed inter language specification, which allows interoperability with external applications. The most basic interoperability feature in the J2EE specification is called JavaIDL. It allows applications to access any CORBA object, written in any language, using the standard IIOP protocol. Due to the J2EE security restrictions, all J2EE application component types should only act as CORBA clients. The only exception is the J2EE application client.

Figure 8-4 shows the position of CORBA in the J2EE architecture.

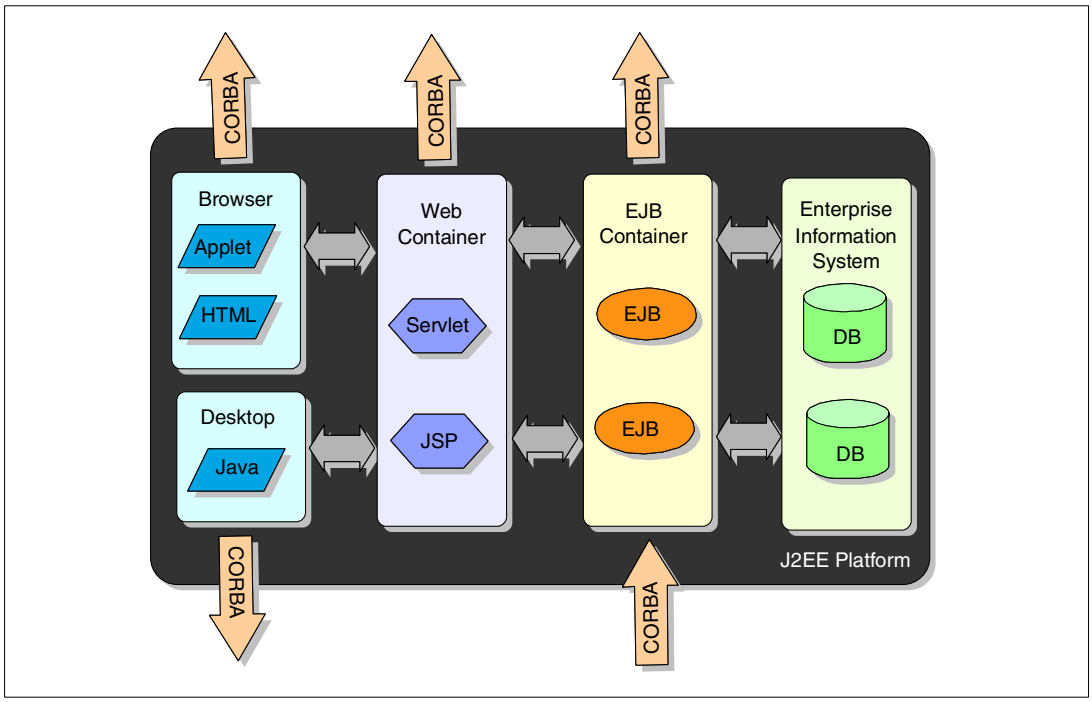


Figure 8-4 Position of CORBA in the J2EE architecture

The J2EE specification version 1.2 defines how the Java Remote Method Invocation (RMI) protocol is mapped onto the IIOP wire-transfer protocol, referred to as RMI over IIOP (RMI-IIOP). In this version of the specification, the J2EE components don't have to support this protocol.

With J2EE version 1.3, clients of EJBs are required to use the RMI-IIOP APIs and the EJB servers themselves have to be accessible by IIOP. This allows EJBs to act as CORBA objects.

If a CORBA client wants to access an EJB object, it is important to be aware of some restrictions. The Java interface *java.io.Serializable* (implemented by HashMap, Date, File, Integer, String, Throwable, and so on) is mapped to the CORBA valuetypes. The consequences are:

- The client has to support the CORBA valuetypes.
- Each Java serializable to be used in the interface between the client and the server (for example, as parameter or return value) has to be re-implemented in the language of the client because they are pass-by-value.

The implementation of Java serializables as value types in C++ or another language can require a significant development effort, as you can see in "Building the CustomerKey valuetype" on page 174.

Note: Valuetypes were introduced by the CORBA 2.3 specification. Many third-party ORBs don't yet implement this specification or don't implement it fully.

You can find more information about the J2EE platform specification versions 1.2 and 1.3 on the Web at: http://java.sun.com/j2ee

CORBA development for EJBs

If the EJB is in the role of the CORBA client, the process is exactly the same as described in "CORBA development" on page 156. If EJB is in the role of the CORBA server, the programming model is different from the standard CORBA programming model. The IDL is not the starting point for the development process, but derived from the EJB classes. The steps are:

1. Develop your EJB just as you would without CORBA.
2. Create the IDL from the EJB and the EJB home classes with a reverse IDL compiler.
3. Compile the IDL files with an IDL compiler for the client stub.
4. Develop the client program:
   a. Develop the main code that invokes the EJB methods.
   b. If serializables are part of the EJB Interface, they have to be implemented as CORBA valuetypes in the client language.
5. Build the client with a language-specific compiler, linker, and archiver.

8.2 Overview of WebSphere Application Server Enterprise Edition CORBA support

The CORBA support in WebSphere Application Server Enterprise Edition is the implementation of the J2EE CORBA services and supports CORBA 2.1, as well as valuetypes from CORBA 2.3. In addition, CORBA support includes the proven interoperability with major CORBA products of other vendors and the implementation of many basic valuetypes for Java serializables (see 8.2.3, “C++ CORBA software development kit (SDK)” on page 162).

8.2.1 WebSphere Application Server support scenarios

If the CORBA specifications were complete, unambiguous, and the vendors had the correct implementations of the same level of the specification, then interoperability would no longer be a concern. However, as already mentioned, this is not the case.

As a result, we need to consider several different scenarios that can be divided into three groups:

- WebSphere Application Server to WebSphere Application Server
- WebSphere Application Server to third-party CORBA interoperation
- WebSphere Application Server to third-party coexistence

WebSphere Application Server to WebSphere Application Server

Only WebSphere Application Server components are involved, so that they are normal J2EE and CORBA business. The following scenarios are possible:

- WebSphere Application Server C++ CORBA client to a WebSphere Application Server EJB server
- WebSphere Application Server C++ CORBA client to a WebSphere Application Server C++ CORBA server
- WebSphere Application Server EJB server (as a CORBA client) to a WebSphere Application Server C++ CORBA server

WebSphere Application Server to third-party CORBA interoperation

WebSphere Application Server interacts with a third-party CORBA ORB. You have to consider the following scenarios:

- Third-party C++ CORBA client to WebSphere Application Server EJB server
- WebSphere Application Server EJB server (as CORBA client) to a third-party CORBA ORB

IBM supports interoperability with its CORBA products for various scenarios. These support statements are the result of controlled tests in specifically identified environments and focus around the IIOP interoperability. The supported products are listed in Table 8-1.

Table 8-1 Client/Server ORBs that interoperate with WebSphere Application Server 4.0

	Orbix		VisiBroker	
	Client	**Server**	**Client**	**Server**
Windows NT **Windows 2000**	3.0.1 C++	3.0.1 C++ Web 3.2 Java 2000 1.2 C++	3.3.3 C++ 4.0 C++ 4.1 C++	3.3.3 C++ 3.4 Java 4.0 C++ 4.0 Java 4.1 C++
AIX 4.3		3.0.1 C++ Web 3.2 Java 2000 1.2 C++	3.3.3 C++ 4.1 C++	3.3.3 C++ 3.4 Java 4.0 Java 4.1 C++
Solaris 7	3.0.2 C++	3.0.2 C++ Web 3.2 Java 2000 1.2 C++	4.0 C++ 4.1 C++	3.4 Java 4.0 C++ 4.0 Java 4.1 C++

The tests that demonstrate interoperability are shipped with WebSphere Application Server as the interoperable sample code. You can run these tests to see, which data types exactly interoperate and which do not. They can be used to verify that ORBs, other than those supported by IBM, interoperate. Note that verifying interoperability with the sample code is not a statement of support by IBM for that ORB.

For our example clients and server, we used newer versions of VisiBroker and Orbix 2000, other than the supported ones, and it still worked.

WebSphere Application Server to third-party coexistence

If a WebSphere Application Server to third-party interoperation scenario is inadequate, you can consider ORB coexistence as an alternative solution. Coexistence is the ability of two different ORB runtime environments to reside and function in the same process.

This can cause problems because the CORBA interfaces were not designed to support this scenario. Only newer ORB runtimes that use the Java ORB portability interfaces can coexist, but even for them, some incompatibilities are possible.

The advantage with coexistence is that the client interoperates with the server (using the ORBs from the same vendor) at all levels: IIOP, services, and proprietary features. Besides, from the possible incompatibilities, this scenario is standard CORBA business.

The ORB coexistence is a feature that has been available since WebSphere Application Server 3.5. Table 8-2 lists the client ORBs that can coexist with WebSphere Application Server 4.0.

Table 8-2 Client ORBs that coexist with WebSphere Application Server 4.0

Platform	Orbix	VisiBroker
Windows NT **Windows 2000**	Web 3.2 Java	3.4 Java 4.0 Java 4.1 Java
AIX 4.3	Web 3.2 Java	3.4 Java 4.0 Java 4.1 Java

Platform	Orbix	VisiBroker
Solaris 7	Web 3.2 Java	3.4 Java 4.0 Java 4.1 Java

For further information, see the WebSphere InfoCenter (navigate to **Enterprise services-> Corba support**) at:
http://www.ibm.com/software/webservers/appserv/doc/v40/aee/index.html

8.2.2 Interoperability with an EJB

The most common way of interoperation with WebSphere Application Server involves EJBs. Therefore, it is worth of taking a closer look at the following two scenarios:

- EJB as a CORBA client
- EJB as a CORBA server

EJB as a CORBA client

If the EJB is the CORBA client, the binding works as described in Figure 8-2 on page 155.

EJB as a CORBA server

The EJB server provides the servant objects (the EJBs) that client applications need to access. In this scenario the following sequence of events occur:

1. The client ORB has to be initialized, when its process is started.
2. The client locates the home interface object of the EJB through the naming service.
3. The home interface object locates the EJB (or creates it, if it doesn't exist).
4. The client creates the stub.
5. The client sends its request through the stub to the EJB.

Figure 8-5 shows the involved components.

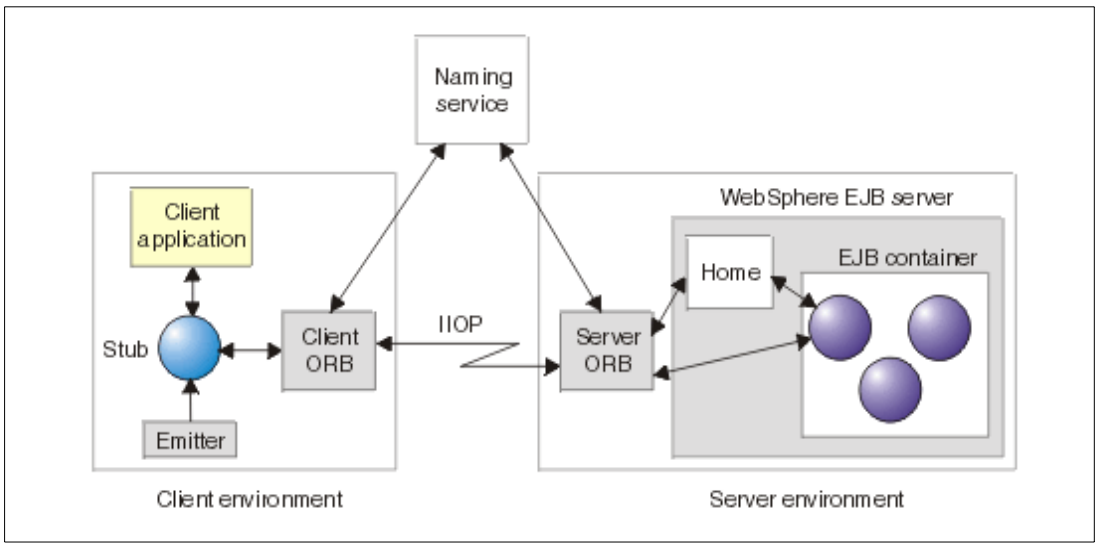


Figure 8-5 An EJB as CORBA server

Currently, the access of a C++ client to an EJB is slightly restricted in WebSphere Application Server 4.0. Session beans are fully supported. For entity beans, only a subset of the home interface methods is supported.

This restriction is derived from the assumption that session beans contain the majority of the business logic and coordinate access to the entity beans (which mostly just provide an object oriented wrapper for a database table or other persistent store). If you follow the best practices and implement your business logic in the session beans, you won't need to access an entity bean directly.

The support for entity beans includes:

- Creating a bean
- Finding a single bean
- Calling a business method
- Using client coordinated transactions (for accessing TX_MANDATORY entity beans)

In general, the EJB specification 1.1 allows you to return either a single object of your bean class or a collection. The restriction is that you are only able to find single objects. You cannot return the collections.

Since session beans don't have finder methods, this restriction doesn't apply to them.

8.2.3 C++ CORBA software development kit (SDK)

The C++ CORBA SDK is a toolkit that allows you to create the WebSphere Application Server CORBA C++ clients and servers that can support interoperability as described in 8.2.1, "WebSphere Application Server support scenarios" on page 159. The SDK conforms to the CORBA 2.1 specification.

The WebSphere Application Server ORB runtime is a collection of several libraries and processes. Some of the processes are C++ daemons, and others are implemented in Java as part of the WebSphere Administration Server JVM. To have the ORB runtime completely initialized, the AdminServer has to be started.

Note: To install C++ SDK, you have to select **Custom** install and make sure the **C++ SDK** box is selected.

The naming, location, and transaction services are implemented in Java. The implementation repository is done as a C++ daemon, regimpl.exe. It is located in the directory %WAS_HOME%\Enterprise\bin. In the implementation repository, you have to register your C++ server. It stores basically a symbolic name for your server, the name of the executable, and the protocol it uses.

The SDK gives you the tools for the development processes described in the "CORBA development" on page 156 and in "CORBA development for EJBs" on page 158. The most important tools are:

- **`idlc`**: IDL compiler for C++ language mapping
- **`rmic`**: Reverse IDL compiler for Java to IDL mapping
- **`Vtlib.lib`**: The CORBA valuetype library for C++, which covers many commonly used Java serializables in the `java.lang`, `java.io`, `java.util`, and `javax.ejb` packages.

Note: The C++ compiler, linker and an archiver are not part of this SDK. You also need a product like Microsoft Visual C++ 6.0 to have a complete development environment.

8.3 C++ CORBA clients for the OrderEntry application

In these scenarios, we have written two different C++ clients for the already existing customer EJB of the OrderEntry application. The clients have restricted access to the methods of the home interface (see “EJB as a CORBA server” on page 161) and full access to the remote interface of the CustomerBean. The clients can:

- Create a customer
- Delete a customer
- Get detail information of a customer
- Set detail information for a customer
- Find a customer by a customer key object

You could imagine another legacy application that needs to get or update customer information. The development process for both clients, in general, follows the steps that are described in “CORBA development for EJBs” on page 158.

8.3.1 Configuring the environment

We used the following environment for our sample clients:

- Windows 2000, SP 1
- WebSphere Application Server V4.0.1 Enterprise Edition
- Microsoft Visual C++ 6.0
- Inprise/Borland VisiBroker 4.5

The installation and configuration of Windows, WebSphere Application Server, and Microsoft Visual C++ are not covered in this redbook. These are described in the product-specific installation guides.

The subdirectory VC98 of the MS Visual C++ installation is referred to as %MSVCDir% in the following sections.

Prerequisites

The following general prerequisites must be fulfilled:

1. Extract the ZIP file containing the CORBA sample code a directory.
2. Set the environment variable OE_HOME to the directory where the ZIP file was extracted.
3. The build process for the IBM client expects the OrderEntry.ear in the directory %OE_HOME%.
4. Microsoft Visual C++ must be installed in the default directory C:\Program Files\Microsoft Visual Studio\VC98. If it is not, you have to edit the environment file %OE_HOME%\corba\set_common_env.bat.

Configuring VisiBroker

For this example, we used an 30-day evaluation version of VisiBroker V4.5. The following setup is described based on this experience. The details may slightly differ if you use a full version of the product.

The VisiBroker 4.5 online documentation (packaged together as HTML books) for C++ and Java is available from the following Web page:
http://www.borland.com/techpubs/books/vbcpp/vbcpp45/framesetindex.html

To install VisiBroker 4.5, you must complete these steps:

1. Install JDK 1.3 from the Web at: `http://java.sun.com/j2se/1.3/download-windows.html`
2. Download the software and register by following the steps at: `http://www.borland.com/visibroker/download/vbcpp4x_steps.html`
3. Install the product as described in the installation guide.
4. Add the environment variable *VBROKERDIR*, which should point to the directory, where VisiBroker is installed.
5. Add %VBROKERDIR%\bin to the PATH environment variable.

For administration purposes, a GUI is shipped with VisiBroker. You start it from the Windows menu bar by selecting **Programs-> VisiBroker-> VisiBroker Console**. Figure 8-6 shows the VisiBroker Console.

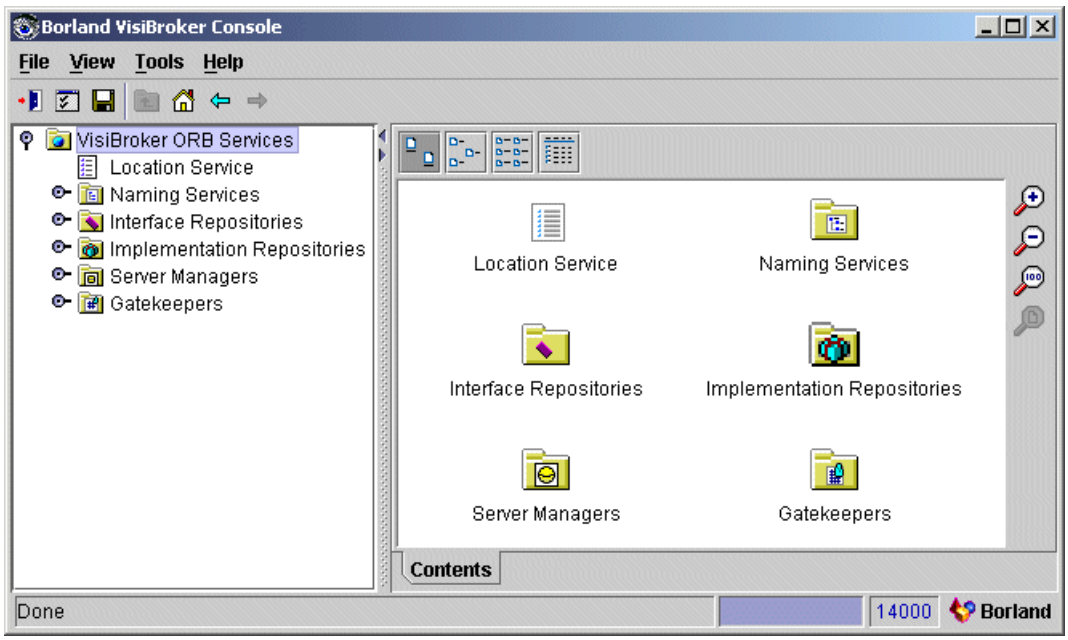


Figure 8-6 VisiBroker Console

It is a browser for the ORB services. The smart agent, which is the basic locator process in VisiBroker and other services, can be started from the windows menu bar via Programs-> VisiBroker or from the command line.

The VisiBroker Console and the administration of the VisiBroker services are described in detail in the programmer's guide in "Part IV Configuration and management".

8.3.2 IBM client of the Customer EJB

The IBM client provides full access to the `CustomerBean` and shows the implementation of a self-created valuetype – `CustomerKey`. The latter one is used in one of the *finder* methods of `CustomerHome`.

The example for the WebSphere Application Server C++ client is located in the directory %OE_HOME%\corba\client\ibm2ejb.

The implementation of this example is based on the “CORBA C++ SDK” sample called “C++ Client that uses valuetypes sample”, which is shipped with WebSphere Application Server. You find the description of all examples in the file:///%WAS_HOME%/Enterprise/samples/index.htm file.

We go through the following basic steps, which are discussed as subsections:

1. Set up the environment.
2. Build the client.
3. Run the client.
4. Build the CustomerKey valuetype.
5. Use the CustomerKey valuetype.

The last two subsections discuss how to create your own valuetype. They are separated from the “Build the client” and “Run the client” because of the complexity and importance of these tasks.

Setting up the environment

To set up the needed environment variables for building and running the client, you have to start a command prompt and execute:

```
%OE_HOME%\corba\client\ibm2ejb\set_ibm2ejb_env.bat
```

Before you start the client, you have to edit the following properties file that sets parameters for the WebSphere Application Server ORB runtime:

```
%OE_HOME%\corba\client\ibm2ejb\ibm2ejb.props
```

The value for `nameServerHost` must match your system's name (including domain name):

```
com.ibm.CORBA.nameServerHost=<hostname>
```

The settings in this property file assume that your system is set up with American English as the default language. If you are using a different default language, you may have to use different values for the settings. For further details, see the documentation of the “C++ Client that uses valuetypes sample”.

Building the client

To build the client, you simply have to run a makefile by executing the following two commands:

```
cd %OE_HOME%\corba\client\ibm2ejb
nmake /f makefile.ibm2ejb
```

This execute all the steps that are described in the subsections of this section:

1. Create the IDL files.
2. Compile the IDL files.
3. Compile and link the stubs.
4. Build the tools library.
5. Build the main object and link the executable.

We show the output of the commands that are executed by the makefile. You don’t have to execute any of the commands manually. But you could do so if you are interested in building the client manually step by step.

The **`nmake`** command is described in detail in the online documentation of Microsoft Visual C++.

nmake may execute the commands in a different order because it doesn't follow a sequence of commands, but resolves dependencies.

To delete all the generated files and directories, you call the two commands:

```
cd %OE_HOME%\corba\client\ibm2ejb
nmake /f makefile.ibm2ejb clean
```

Creating the IDL files

The CORBA interfaces are always described in IDL notation. They are derived from the class files of the `CustomerBean`. The creation of the IDL files is done with a separate makefile, called `makefile.interface`. This makefile is called by the main `makefile.ibm2ejb` and executes the following steps:

1. Extract the classes from the EAR file, which produces the output:

```
*** "Extracting the EJB jar file from the ear file"
        jar -xvf %OE_HOME%\Order_Entry_Application.ear Deployed_OrderEntryEJB.jar
extracted: Deployed_OrderEntryEJB.jar

*** "Extracting the Customer class from the EJB jar file"
        jar -xvf Deployed_OrderEntryEJB.jar com\ibm\itso\roch\wasaejb\Customer.class
extracted: com/ibm/itso/roch/wasaejb/Customer.class

*** "Extracting the CustomerKey class from the EJB jar file"
        jar -xvf Deployed_OrderEntryEJB.jar com\ibm\itso\roch\wasaejb\CustomerKey.class
extracted: com/ibm/itso/roch/wasaejb/CustomerKey.class

*** "Extracting the CustomerHome class from the EJB jar file"
        jar -xvf Deployed_OrderEntryEJB.jar com\ibm\itso\roch\wasaejb\CustomerHome.class
extracted: com/ibm/itso/roch/wasaejb/CustomerHome.class
```

2. Create the IDL files. The output is:

```
*** "Creating the Customer IDL"
        rmic -J"-Djava.ext.dirs=%WAS_HOME%\lib;%WAS_HOME%\Enterprise\lib" -idl
com.ibm.itso.roch.wasaejb.Customer

*** "Creating the CustomerKey IDL"
        rmic -J"-Djava.ext.dirs=%WAS_HOME%\lib;%WAS_HOME%\Enterprise\lib" -idl
com.ibm.itso.roch.wasaejb.CustomerKey

*** "Creating the CustomerHome IDL"
        rmic -J"-Djava.ext.dirs=%WAS_HOME%\lib;%WAS_HOME%\Enterprise\lib" -idl
com.ibm.itso.roch.wasaejb.CustomerHome
```

Compiling the IDL files

The client stub files are built from IDL files. For example, they do the data type marshalling and have to be linked to the main executable. `makefile.ibm2ejb` generates the client stubs that produce the following output:

```
*** "Generating the Customer client stub"
      idlc -J"-Djava.ext.dirs=%WAS_HOME%\lib;%WAS_HOME%\Enterprise\lib" -I . -d
com\ibm\itso\roch\wasaejb -mcpponly  -mdllname=Customer -shh:uc
.\com\ibm\itso\roch\wasaejb\Customer.idl

*** "Generating the CustomerKey client stub"
      idlc -J"-Djava.ext.dirs=%WAS_HOME%\lib;%WAS_HOME%\Enterprise\lib" -I . -d
com\ibm\itso\roch\wasaejb -mcpponly  -mdllname=Customer -shh:uc
.\com\ibm\itso\roch\wasaejb\CustomerKey.idl

*** "Generating the CustomerHome client stub"
```

```
      idlc -J"-Djava.ext.dirs=%WAS_HOME%\lib;%WAS_HOME%\Enterprise\lib" -I . -d
com\ibm\itso\roch\wasaejb -mcpponly  -mdllname=Customer -shh:uc
.\com\ibm\itso\roch\wasaejb\CustomerHome.idl
```

Compiling and linking the stubs

The stub files have to be compiled. They could have been directly linked to the main executable, but we decided to put them together in a DLL. Note that the name of the library already is provided in the previous step to the IDL compiler as parameter `-mdllname=Customer`. `makefile.ibm2ejb` executes the following steps:

1. Compile the stubs to object files. The output is:

```
*** "Generating the Customer object"
        cl /I%OE_HOME%\corba\tools /I. /I%WAS_HOME%\Enterprise\include /I\include
-DEXCL_IRTC -DSHASTIFIED -D_USE_NAMESPACE  -GX  /c /nologo /MD /Od -DSOM_DLL_Customer
com\ibm\itso\roch\wasaejb\Customer_C.cpp Customer_C.cpp

*** "Generating the CustomerKey object"
        cl /I%OE_HOME%\corba\tools /I. /IC%WAS_HOME%\Enterprise\ include /I\include
-DEXCL_IRTC -DSHASTIFIED -D_USE_NAMESPACE  -GX  /c /nologo /MD /Od -DSOM_DLL_Customer
com\ibm\itso\roch\wasaejb\CustomerKey_C.cpp
CustomerKey_C.cpp

*** "Generating the CustomerKey valuetype object"
        cl /I%OE_HOME%\corba\tools /I. /I%WAS_HOME%\Enterprise\ include /I\include
-DEXCL_IRTC -DSHASTIFIED -D_USE_NAMESPACE  -GX  /c /nologo /MD /Od -DSOM_DLL_Customer
com\ibm\itso\roch\wasaejb\CustomerKey_I.cpp
CustomerKey_I.cpp

*** "Generating the CustomerHome object"
        cl /I%OE_HOME%\corba\tools /I. /I%WAS_HOME%\Enterprise\ include /I\include
-DEXCL_IRTC -DSHASTIFIED -D_USE_NAMESPACE  -GX  /c /nologo /MD /Od -DSOM_DLL_Customer
com\ibm\itso\roch\wasaejb\CustomerHome_C.cpp
CustomerHome_C.cpp
```

2. Link the stub library and DLL. This produces the following output:

```
*** "Creating the Customer.lib library"
        lib /nologo /LIBPATH:%OE_HOME%\corba\tools /LIBPATH:.
/LIBPATH:%WAS_HOME%\Enterprise\lib  /LIBPATH:\lib /OUT:Customer.lib /def: Customer_C.obj
CustomerKey_C.obj CustomerKey_I.obj CustomerHome_C.obj
   Creating library Customer.lib and object Customer.exp

*** "Linking the Customer.dll library"
        link /NOLOGO /INCREMENTAL:NO  /DEBUG /LIBPATH:%OE_HOME%\corba\tools /LIBPATH:.
/LIBPATH:%WAS_HOME%\Enterprise\lib  /LIBPATH:\lib somororm.lib  somosa1m.lib Vtlib.lib
kernel32.lib user32.lib /DLL Customer.exp Customer_C.obj CustomerKey_C.obj
CustomerKey_I.obj CustomerHome_C.obj /OUT:Customer.dll
```

Note that the `CustomerKey` valuetype class has been compiled and linked as well in the last two steps. The description of how to create this class is provided in “Building the CustomerKey valuetype” on page 174.

Building the tools library

The tools library consists of some classes that are used from both client implementations. These classes provide trace and error handling utilities and include the `MyOrb` class, which encapsulates the access to the ORB. Also it includes two wrapper classes for the `Customer` and the `CustomerHome` class, which hide the CORBA handling. The creation of the tools library is done with a separate makefile, called `makefile.tools`. It inherits its parameter settings from `makefile.ibm2ejb`. `makefile.tools` is called by `makefile.ibm2ejb`. It executes the following steps:

1. Compile the tools classes. The output is:

```
   set INCLUDES=/I%OE_HOME%\corba\client\ibm2ejb /I%OE_HOME%\corba\tools
/I%WAS_HOME%\Enterprise\include /I\include
   set CC_FLAGS=/GX /c /nologo /MD /Od /DEXCL_IRTC /DSHASTIFIED /D_USE_NAMESPACE
/DUSE_JAVA_EX /DORB_WebSphere
   set LIBS=/LIBPATH:%OE_HOME%\corba\client\ibm2ejb /LIBPATH:%OE_HOME%\corba\tools
/LIBPATH:%WAS_HOME%\Enterprise\lib  /LIBPATH:%MSVCDir%\lib
   set LD_FLAGS=/NOLOGO /INCREMENTAL:NO /DEBUG /LIBPATH:%OE_HOME%\corba\client\ibm2ejb
/LIBPATH:%OE_HOME%\corba\tools /LIBPATH:%WAS_HOME%\Enterprise\lib
/LIBPATH:%MSVCDir%\lib somororm.lib  somosa1m.lib Vtlib.lib kernel32.lib user32.lib
   set LIB_FLAGS=/nologo
   cd %OE_HOME%\corba\tools
   nmake -nologo -f makefile.tools

*** "Compiling the Trace class"
   cl /I%OE_HOME%\corba\client\ibm2ejb /I%OE_HOME%\corba\tools
/I%WAS_HOME%\Enterprise\include /I\include /GX /c /nologo /MD /Od /DEXCL_IRTC
/DSHASTIFIED /D_USE_NAMESPACE  /DUSE_JAVA_EX /DORB_WebSphere Trace.cpp
Trace.cpp

*** "Compiling the Error class"
   cl /I%OE_HOME%\corba\client\ibm2ejb /I%OE_HOME%\corba\tools
/I%WAS_HOME%\Enterprise\include /I\include /GX /c /nologo /MD /Od /DEXCL_IRTC
/DSHASTIFIED /D_USE_NAMESPACE  /DUSE_JAVA_EX /DORB_WebSphere Error.cpp
Error.cpp

*** "Compiling the Warning class"
   cl /I%OE_HOME%\corba\client\ibm2ejb /I%OE_HOME%\corba\tools
/I%WAS_HOME%\Enterprise\include /I\include /GX /c /nologo /MD /Od /DEXCL_IRTC
/DSHASTIFIED /D_USE_NAMESPACE  /DUSE_JAVA_EX /DORB_WebSphere Warning.cpp
Warning.cpp

*** "Compiling the MyOrb class"
   cl /I%OE_HOME%\corba\client\ibm2ejb /I%OE_HOME%\corba\tools
/I%WAS_HOME%\Enterprise\include /I\include /GX /c /nologo /MD /Od /DEXCL_IRTC
/DSHASTIFIED /D_USE_NAMESPACE  /DUSE_JAVA_EX /DORB_WebSphere MyOrb.cpp
MyOrb.cpp

*** "Compiling the CustomerWrapper class"
   cl /I%OE_HOME%\corba\client\ibm2ejb /I%OE_HOME%\corba\tools
/I%WAS_HOME%\Enterprise\include /I\include /GX /c /nologo /MD /Od /DEXCL_IRTC
/DSHASTIFIED /D_USE_NAMESPACE  /DUSE_JAVA_EX /DORB_WebSphere CustomerWrapper.cpp
CustomerWrapper.cpp

*** "Compiling the CustomerFactory class"
   cl /IF:\Henning\oe\corba\client\ibm2ejb /IF:\Henning\oe\corba\tools
/I%WAS_HOME%\Enterprise\include /I\include /GX /c /nologo /MD /Od /DEXCL_IRTC
/DSHASTIFIED /D_USE_NAMESPACE  /DUSE_JAVA_EX /DORB_WebSphere CustomerFactory.cpp
CustomerFactory.cpp
```

2. Link the tools library and DLL, which produce the output:

```
*** "Creating the tools lib"
   lib /nologo /LIBPATH:. /LIBPATH:%WAS_HOME%\Enterprise\lib /LIBPATH:\lib
/OUT:tools.lib /def: Trace.obj Error.obj Warning.obj MyOrb.obj
   Creating library tools.lib and object tools.exp

*** "Linking the tools.dll library"
   link /NOLOGO /INCREMENTAL:NO  /DEBUG /LIBPATH:. /LIBPATH:%WAS_HOME%\Enterprise\lib
/LIBPATH:\lib somororm.lib  somosa1m.lib Vtlib.lib kernel32.lib user32.lib /DLL
tools.exp Trace.obj Error.obj Warning.obj MyOrb.obj /OUT:tools.dll
```

```
cd %OE_HOME%\corba\client\ibm2ejb
move %OE_HOME%\corba\tools\tools.dll .
```

Building the main object and linking the executable

The *main()* method is located in the ibm2ejb.cpp file. Beside the tools and stubs libraries, it uses the `ProgVals` class to read the program arguments. The `makefile.ibm2ejb` file executes the following final steps:

1. Compile the main and the `ProgVals` object. The output is:

```
*** "Generating the ibm2ejb.obj object"
   cl /I%OE_HOME%\corba\client\ibm2ejb /I%OE_HOME%\corba\tools
/I%WAS_HOME%\Enterprise\include /I\include /GX /c /nologo /MD /Od /DEXCL_IRTC
/DSHASTIFIED /D_USE_NAMESPACE  /DUSE_JAVA_EX /DORB_WebSphere ibm2ejb.cpp
ibm2ejb.cpp

*** "Compiling the ProgVals class"
   cl /I%OE_HOME%\corba\client\ibm2ejb /I%OE_HOME%\corba\tools
/I%WAS_HOME%\Enterprise\include /I\include /GX /c /nologo /MD /Od /DEXCL_IRTC
/DSHASTIFIED /D_USE_NAMESPACE  /DUSE_JAVA_EX /DORB_WebSphere ProgVals.cpp
ProgVals.cpp
```

2. Link the executable, which produces the output:

```
*** "Linking the ibm2ejb.exe executable"
   link /NOLOGO /INCREMENTAL:NO /DEBUG /LIBPATH:%OE_HOME%\corba\client\ibm2ejb
/LIBPATH:%OE_HOME%\corba\tools /LIBPATH:%WAS_HOME%\Enterprise\lib
/LIBPATH:%MSVCDir%\lib somororm.lib  somosa1m.lib Vtlib.lib kernel32.lib user32.lib
ibm2ejb.obj ProgVals.obj  Customer.lib F:\Henning\oe\corba\tools\tools.lib
/SUBSYSTEM:CONSOLE /OUT:ibm2ejb.exe
   Creating library ibm2ejb.lib and object ibm2ejb.exp
```

The output of the whole process is the executable file ibm2ejb.exe.

Running the client

To run the client, you should execute the following command. The general usage of the executable is:

```
ibm2ejb method=<method> [id=<id>] [forename=<fname>] [lastname=<lname>] [phone=<phone>]
```

The following methods are available:

- create [id=<id>] [forename=<fname>] [lastname=<lname>] [phone=<phone>]
- remove [id=<id>]
- getPhone [id=<id>]
- setPhone [phone=<phone>]
- findByPrimaryKey [id=<id>]

To create a customer with default values, you call:

```
ibm2ejb method=create id=1234 forename=joe lastname=miller phone=01234-56789
```

The output of the executable would be like the following example:

```
TRACE: Entering main ...
TRACE: Reading program arguments ...
TRACE: Program values:
TRACE: forename=joe, id=1234, lastname=miller, method=create, phone=01234-56789,
TRACE: Initializing the ORB ...
TRACE: By bootstrapping ...
TRACE: Getting name server ...
TRACE: Constructing name entry ...
TRACE: Resolving name entry domain/legacyRoot/CustomerGlobal ...
```

```
TRACE: Narrowing CustomerHome ...
TRACE: Creating a CustomerBean with the following parameters:
TRACE: CustomerID: 1234
TRACE: ForeName:   joe
TRACE: LastName:   miller
TRACE: Phone:      01234-56789
TRACE: Customer created
TRACE: ( 1234, joe             , miller          , 01234-56789     )
```

Important: At the time this redbook was written, a call failed to the *findByPrimaryKey* method with the key class that wraps `java.lang.String`. The problem was reported to the WebSphere development team.

However, this problem needs more clarification. EJB specification v1.1 recommends using the EJB's primary key *field* as the primary key *class*, when a primary key class maps to a single field. Following this recommendation, we would need to use `java.lang.String` (customer ID filed) in the *findByPrimaryKey()* method. WebSphere Application Server works fine with `String`.

The following subsections explain what happens when ibm2ejb.exe is executed:

- Class overview
- Initializing the IBM ORB
- Getting the root context by bootstrapping
- Binding the CustomerHome
- Creating a customer

Class overview

Figure 8-7 shows an overview of the involved classes and how they are related. The classes `Customer`, `CustomerHome`, and `CustomerKey`, are generated. Therefore, they are displayed in a different style.

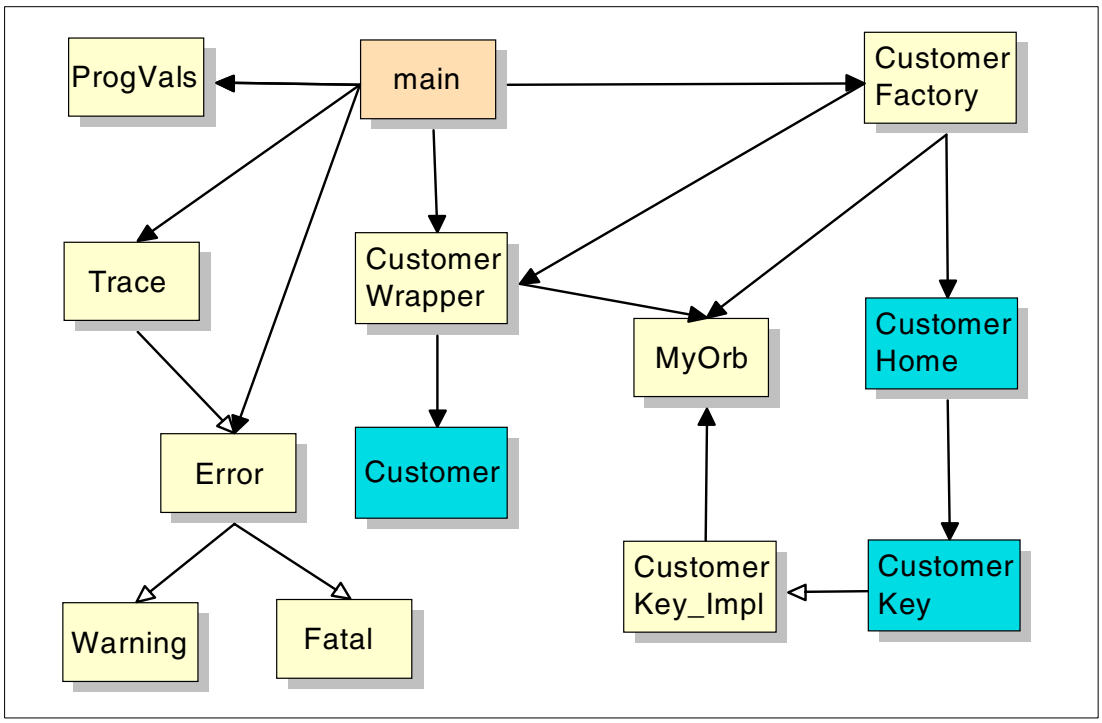


Figure 8-7 Class diagram for ibm2ejb

You can divide the classes into two groups:

- IDL generated classes:
  - Customer
  - CustomerHome
  - CustomerKey
- Manually created classes:
  - main
  - CustomerWrapper
  - CustomerFactory
  - CustomerKey_Impl
  - MyOrb
  - ProgVals
  - Trace
  - Error
  - Warning
  - Fatal

You find the source code of the described classes in the directories:

```
%OE_HOME%\corba\client\ibm2ejb
%OE_HOME%\corba\client\ibm2ejb\com\ibm\itso\roch\wasaejb
%OE_HOME%\corba\tools
```

Initializing the IBM ORB

After the *main* method has read the program parameters, the ORB is initialized. This is done inside the constructor of the `MyOrb` class. Example 8-1 shows the code.

Example 8-1 Initializing the IBM ORB

```
::CORBA::ORB_ptr _orb;

void MyOrb::initOrb()
{
...
   int int0 = 0;

   //Initialize the ORB
   try {
      t.println("Initializing the ORB ...");
      _orb = CORBA::ORB_init(int0, NULL, "DSOM");
   }
...
}
```

Getting the root context by bootstrapping

The next step is to get the root context of the naming service. Example 8-2 shows the method that handles this task. First the initial reference of the NameService is resolved by bootstrapping and then narrowed to the `NamingContext` object.

Example 8-2 Getting root context by bootstrapping

```
::CORBA::ORB_ptr MyOrb::_orb;
::std::string MyOrb::_rootIOR;
```

```
::CosNaming::NamingContext_ptr MyOrb::getRootContext()
{
...
   ::CosNaming::NamingContext_ptr rootContext = NULL;
...
   try {
      t.println("Getting name server ...");
      ::CORBA::Object_var obj;
      if (_rootIOR == "") {
         t.println("By bootstrapping ...");
         obj = _orb->resolve_initial_references("NameService");
      }
...
      if (CORBA::is_nil(obj)) {
         err.println("resolve_initial_references(\"NameService\") returned nil\nMake sure
your server is running and you set SOMCBPROPS!");
      }
      rootContext = ::CosNaming::NamingContext::_narrow(obj);
      if (CORBA::is_nil(rootContext)) {
         err.println("name server narrowed to nil");
      }
   }
...
   return rootContext;
}
```

Binding CustomerHome

The connection to the `CustomerHome` is established through the `CustomerFactory`. It is a wrapper class that offers the methods of the `CustomerHome` and handles the possible remote exceptions. The *create* and *find* methods of the `CustomerFactory` don't return `Customer` objects, but `CustomerWrapper` objects, which do the exception handling for the `Customer` calls.

Example 8-3 shows the code fragments that create the binding to the `CustomerHome`. The first step is to create a name structure containing the link to the entry of the `CustomerHome`. The next step is to resolve this name by the naming service. In the last step, the resolved CORBA object is narrowed to an `CustomerHome` object.

Example 8-3 Binding the CustomerHome

```
const string CustomerFactory::NS_ENTRY = "domain/legacyRoot/CustomerGlobal";
::com::ibm::itso::roch::wasaejb::CustomerHome_ptr CustomerFactory::_customerHome;

CustomerFactory::CustomerFactory(const ::CosNaming::NamingContext_ptr rootContext)
{
...
   // get naming entry in Name structure format
   t.println("Constructing name entry ...");
   ::CosNaming::Name *name = MyOrb::stringToName(NS_ENTRY);
...
      ::CORBA::Object_var homeObj = NULL;
      t.println("Resolving name entry " + NS_ENTRY + " ...");
      homeObj = rootContext->resolve(*name);
      if (CORBA::is_nil(homeObj)) {
         err.println("apparently resolve FAILED! ");
      }
...
      t.println("Narrowing CustomerHome ...");
      _customerHome = CustomerHome::_narrow(homeObj);
```

```
      if (CORBA::is_nil(_customerHome)) {
         err.println("narrow returned nil on CustomerHome");
      }
...
}

::CosNaming::Name* MyOrb::stringToName(const string nameString)
{
   Trace t;
   char sep = '/';
   string token;
   ::CosNaming::Name *name = new ::CosNaming::Name();

   // count the number of parts
   int i;
   int partNo = 1;
   for (i=nameString.find(sep); i!=nameString.npos; i=nameString.find(sep, i+1))
      partNo++;
   name->length(partNo);

   int start = 0;
   int end, j;
   for (j=0; j<partNo; j++) {
      end   = nameString.find(sep, start);
      token = nameString.substr(start, end - start);
      //t.println("found token: " + token);

      (*name)[j].id   = ::CORBA::string_dup(token.c_str());
      (*name)[j].kind = ::CORBA::string_dup("");
      start = end + 1;
   }

   return name;
}
```

Creating a customer

After `CustomerHome` is instantiated, all methods of the `CustomerHome` can be used. This section covers the creation of a Customer object. The other calls are similar.

Example 8-4 shows the important parts of the `CustomerFactory`'s *createBean()* method.

Example 8-4 Creating a customer

```
CustomerWrapper CustomerFactory::createBean(string customerID, string foreName, string
lastName, string phone)
{
...
   Customer_var bean = NULL;
...
      bean = _customerHome->create__CORBA_WStringValue__CORBA_WStringValue__CORBA
_WStringValue__CORBA_WStringValue(
         MyOrb::stringToWStringValue(customerID.c_str()),
         MyOrb::stringToWStringValue(foreName.c_str()),
         MyOrb::stringToWStringValue(lastName.c_str()),
         MyOrb::stringToWStringValue(phone.c_str()) );
      if (CORBA::is_nil(bean))
         err.println("Create returned nil");
...
```

```
    return CustomerWrapper(bean);
}

char* MyOrb::wstringValueToString(::CORBA::WStringValue_ptr wsv)
{
    return ::com::ibm::ws::VtlUtil::WStringValueToString(wsv);
}
```

Building the CustomerKey valuetype

The *findByPrimaryKey()* method of `CustomerHome` takes, as a parameter, an object of the type `CustomerKey`. Since this is self-defined, it cannot be part of the IBM valuetype library. So to use this method, we have to create our own C++ valuetype of that class.

This means that we have to re-implement in C++ the `CustomerKey` class with its methods, that are described in the interface *`CustomerKey.hh`*. Our example already includes the final version of the implementation files. We describe the steps that we have gone through. These are the steps to create the `CustomerKey` valuetype:

1. Generate the skeleton files from the IDL by running these two commands:

   ```
   cd %OE_HOME%\corba\client\ibm2ejb
   idlc -J"-Djava.ext.dirs=%WAS_HOME%\lib;%WAS_HOME%\Enterprise\lib" -I . -d
   com\ibm\itso\roch\wasaejb -mcpponly -mdllname=Customer -eih:ic
   .\com\ibm\itso\roch\wasaejb\CustomerKey.idl
   ```

2. Edit the files produced in the previous step:

   ```
   CustomerKey.ih
   CustomerKey_I.cpp
   ```

3. Add the following inheritance to the CustomerKey.ih file to use the default implementation class. The following line:

   ```
   class com_ibm_itso_roch_wasaejb_CustomerKey_Impl
   : virtual public ::com::ibm::itso::roch::wasaejb::CustomerKey,
   virtual public ::CORBA::DefaultValueRefCountBase
   ```

   was changed to:

   ```
   class com_ibm_itso_roch_wasaejb_CustomerKey_Impl
   : virtual public ::com::ibm::itso::roch::wasaejb::CustomerKey,
   virtual public OBV_com::ibm::itso::roch::wasaejb::CustomerKey,
   virtual public ::CORBA::DefaultValueRefCountBase
   ```

4. Implement the two methods of the `CustomerKey` class in the CustomerKey_I.cpp file:

   ```
   ::CORBA::Boolean com_ibm_itso_roch_wasaejb_CustomerKey_Impl::equals (const
   ::java::lang::Object & obj)
   {
      cout << "Entering CustomerKey_Impl::equals()" << endl;

      ::CORBA::Boolean areEqual = FALSE;
      ::CORBA::ValueBase* tmp = (::CORBA::ValueBase*)(&obj);
      CustomerKey_var ckey = CustomerKey::_narrow(tmp);
      if (ckey != (::CORBA::ValueBase*)(NULL)) {
         char* keyVal1 = VtlUtil::WStringValueToString(ckey->primaryKey());
         char* keyVal2 = VtlUtil::WStringValueToString(primaryKey());
         areEqual = (::strcmp(keyVal1, keyVal2) == 0);
      }

      return areEqual;
   }
   ```

Note: The getters and setters of `OBV_com::ibm::itso::roch::wasaejb::CustomerKey` have to be used to access the member primaryKey.

```
::CORBA::Long com_ibm_itso_roch_wasaejb_CustomerKey_Impl::hashCode ()
{
   ::CORBA::Long hash = 0;

   ::CORBA::Long prime = 31;
   char* key = VtlUtil::WStringValueToString(primaryKey());
   try {
      for (::CORBA::Long i = 0; i < ::strlen(key) ; i++)
         hash = prime*hash + key[i];
   } catch (...) {
      cerr << "Caught an unexpected exception in CustomerKey::hashCode()" << endl;
   }

   return hash;
}
```

Note: The algorithm for the *hashCode()* method was taken from the *java.lang.string.hashCode()* implementation in IBM's JDK 1.2.2.

5. The `CustomerKey` objects are created by a factory. This `CustomerKey_factory` class has to be defined in *`CustomerKey.ih`*. Add the factory class with the two creation and two standard methods:

```
class com_ibm_itso_roch_wasaejb_CustomerKey_factory : public
::com::ibm::itso::roch::wasaejb::CustomerKey_init
{
public:
   // create methods of the CustomerKey_init class in CustomerKey.hh
   virtual ::com::ibm::itso::roch::wasaejb::CustomerKey* create__ ();
   virtual ::com::ibm::itso::roch::wasaejb::CustomerKey* create__CORBA_WStringValue
(::CORBA::WStringValue* key);

   // general methods
   virtual CORBA::ValueBase* create_for_unmarshal();
   ::CORBA::ValueBase* asValueBase(void * v);
};
```

Note: The *create* methods can be copied from the `CustomerKey_init` class in *`CustomerKey.hh`*.

6. Add the implemenations of the factory class methods to `CustomerKey_I.cpp`:

```
CORBA::ValueBase* com_ibm_itso_roch_wasaejb_CustomerKey_factory::create_for_unmarshal()
{
   return new com_ibm_itso_roch_wasaejb_CustomerKey_Impl();
}

CustomerKey* com_ibm_itso_roch_wasaejb_CustomerKey_factory::create__ ()
{
   return new com_ibm_itso_roch_wasaejb_CustomerKey_Impl();
}

CustomerKey* com_ibm_itso_roch_wasaejb_CustomerKey_factory::create__CORBA_WStringValue
(::CORBA::WStringValue* key)
{
```

```
   CustomerKey* ckey = new com_ibm_itso_roch_wasaejb_CustomerKey_Impl();
   ckey->primaryKey(key);
   return ckey;
}

::CORBA::ValueBase* com_ibm_itso_roch_wasaejb_CustomerKey_factory::asValueBase(void * v)
{
    return  (::CORBA::ValueBase *) (::com::ibm::itso::roch::wasaejb::CustomerKey*) v;
}
```

7. Provide the additional class `CustomerKey_factory_init` and instantiate it in `CustomerKey_I.cpp`. This class is responsible for the registration of the factory with the `ORB`:

```
class CustomerKey_factory_init
{
   public:
   CustomerKey_factory_init()
   {
   ::CORBA::ORB_ptr _vtlibOrb;
   int _argc = 0;
   ::CORBA::ValueFactoryBase_var retfact;
   ::CORBA::ValueFactoryBase_var fact;

   // Get access to the ORB.
      _vtlibOrb = ::CORBA::ORB_init(_argc, NULL, "DSOM");

   // Create a CustomerKey factory.
   fact = new com_ibm_itso_roch_wasaejb_CustomerKey_factory();
   retfact = _vtlibOrb->register_value_factory(
      const_cast<char*>(CustomerKey::CustomerKey_RID),
      fact.in() );
}
};

// Static instantiation of the class.
static CustomerKey_factory_init __theCustomerKey_factory_init;
```

For a detailed description about how to build a valuetype, see the WebSphere InfoCenter (navigate to **Enterprise services-> Corba support-> Creating your own C++ valuetypes**) at: http://www.ibm.com/software/webservers/appserv/doc/v40/aee/index.html.

Using the CustomerKey valuetype

The previous section showed how to create the `CustomerKey` valuetype. This section shows how to instantiate and use it. The `CustomerKey` object is needed as a parameter of the `CustomerHome` *findByPrimaryKey()* method.

The creation of `CustomerKey` objects is done in two steps:

1. Instantiate `CustomerKey_factory`. This is done in the constructor of the `CustomerFactory` class, which uses `CustomerKey`:

```
CustomerFactory::CustomerFactory(const ::CosNaming::NamingContext_ptr rootContext)
   :_customerHome(NULL)
    ,_keyFactory( (::com::ibm::itso::roch::wasaejb::CustomerKey_init*) VtlUtil::
getFactory(const_cast<char*>(::com::ibm::itso::roch::wasaejb::CustomerKey::
CustomerKey_RID)) )
```

2. When the factory is created, the `CustomerKey` objects are created by the *create()* methods of the factory:

```
   ::com::ibm::itso::roch::wasaejb::CustomerKey_var ckey =
_keyFactory->create__CORBA_WStringValue(MyOrb::toWStringValue(key.c_str()));
```

8.3.3 VisiBroker client of the Customer EJB

The VisiBroker client provides partial access to `CustomerBean`. It is more work to connect this client, because WebSphere Application Server's valuetype library cannot be used and has to be created manually for the needs of the client.

The example for the VisiBroker C++ client is located in the %OE_HOME%\corba\client\vb2ejb directory. The implementation of this example is based on the "ORBs interoperability" sample called "WebSphere EJB server to VisiBroker 4.1 C++ client", which is shipped with WebSphere Application Server. You find the description of all examples in the file:

```
file:///%WAS_HOME%/Enterprise/samples/index.htm
```

Compiling and running the mentioned example gives a good hint in regard to which data types are supported.

Note: For our sample, we could see that `javax::ejb::CreateException` and `javax::ejb::FinderException` are not supported by VisiBroker 4.5.

We go through the basic steps, which are discussed in the following subsections:

1. Set up environment
2. Build the client
3. Run the client

Setting up environment

To set up the needed environment variables for building and running the client, you have to start a command prompt and execute:

```
%OE_HOME%\corba\client\vb2ejb\set_vb2ejb_env.bat
```

Some settings need to be done on the server side to properly transport the string parameters. The IBM client doesn't need this setting, but it doesn't prevent the IBM client from running.

In the WebSphere Administrative Console, you have to change the JVM settings of the application server that runs the OrderEntry application. You have to add the system property as shown in Figure 8-8.

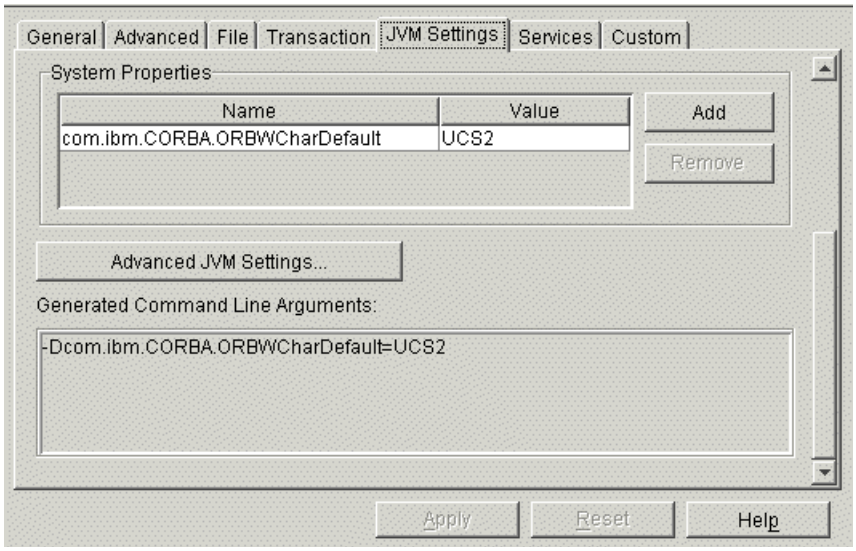


Figure 8-8 JVM settings for VisiBroker

Building the client

To build the client, you simply have to run a makefile by executing the following two commands:

```
cd %OE_HOME%\corba\client\vb2ejb
nmake /f makefile.vb2ejb
```

This command executes all the steps that are described in the following subsections:

1. Create the IDL files
2. Merge the IDL files
3. Compile the IDL files
4. Compile and link the stubs
5. Build the tools library
6. Build the main object and link the executable
7. Build the IOR tool

We show the output of the commands that are executed by the makefile. You don't have to execute any of the commands manually. But you could do so, if you are interested in building the client manually step by step.

The **nmake** command is described in detail in the online documentation of Microsoft Visual C++. The `makefile.ibm2ejb` file executes all the steps, that are described in the following subsections. **nmake** may execute the commands in a different order because it doesn't follow a sequence of commands, but resolves dependencies.

To delete all the generated files and directories, you call:

```
cd %OE_HOME%\corba\client\vb2ejb
nmake /f makefile.vb2ejb clean
```

Creating the IDL files

The IDL files were created in the same way as described in "Creating the IDL files" on page 166. But they have to be edited manually before they can be compiled by idl2cpp (the IDL compiler of VisiBroker). We have to do that because:

- VisiBroker does not have a valuetype library like WebSphere Application Server.
- Some data types are not supported.
- The `idl2cpp` compiler does not support C++ namespaces properly.

To address the first two problems, we had to:

1. Copy the following files from the "WebSphere EJB server to VisiBroker 4.1 C++ client" sample:
   - Vtlib.idl
   - Vtlib_vb.hh
   - Vtlib_I_vb.cpp
2. Delete all includes from the `Customer.idl`, `CustomerKey.idl`, and `CustomerHome.idl` besides:

   ```
   #include "orb.idl"
   ```

3. Add as replacement for the removed references immediately after the include of orb.idl:

   ```
   #include "Vtlib.idl"
   ```

4. Delete the raise of the exceptions `::javax::ejb::FinderEx` and `::javax::ejb::CreateEx` from the method declarations.
5. Remove all Java modules from the three IDL files.

OMG IDL to C++ language mapping specifies that each IDL module should be mapped to a C++ namespace with the same name. If namespaces are not supported by the C++ compiler, the modules can be mapped on classes. But that means, that several IDL files containing a module with the same name will generate C++ header files with different classes having the same name. Therefore, the solution for problem three is to:

1. Merge the three IDL files to one file – Customer_merged.idl.
2. Rename the IDL files to 1_Customer.idl, 2_CustomerKey.idl, and 3_CustomerHome.idl to merge them in the right order.
3. Create two empty files, 1_Customer_c.hh and 2_CustomerKey_c.hh, to include directives from the generated Customer_merged_c.hh.

The output of all these manual activities is the following five files in the directory %OE_HOME%\corba\client\vb2ejb\com\ibm\itso\roch\wasaejb:

- 1_Customer.idl
- 2_CustomerKey.idl
- 3_CustomerHome.idl
- 1_Customer_c.hh
- 2_CustomerKey_c.hh

Merging the IDL files

The **makefile** utility automatically merges the IDL files. The **makefile** utility merges the IDL files. The output is:

```
*** "Merging the Customer* IDLs"
   copy /b .\com\ibm\itso\roch\wasaejb\*_Customer*.idl Customer_merged.idl
.\com\ibm\itso\roch\wasaejb\1_Customer.idl
.\com\ibm\itso\roch\wasaejb\2_CustomerKey.idl
.\com\ibm\itso\roch\wasaejb\3_CustomerHome.idl
        1 file(s) copied.
```

Compiling the IDL files

The client stub files are built from the merged IDL file. In addition, the IDL for the valuetypes has to be compiled. Note that this is a different IDL compiler than the one that was used in “Compiling the IDL files” on page 166. The **makefile** utility generates the client stubs, which produce the output:

```
*** "Generating the Customer_merged client stubs"
   idl2cpp -I%VBROKERDIR%\idl -I. -src_suffix cpp -export __declspec(dllexport)
Customer_merged.idl
*** "Warnings are ignored"

*** "Generating the client stubs of the value type library"
   idl2cpp -I%VBROKERDIR%\idl -I. -src_suffix cpp -export __declspec(dllexport) Vtlib.idl
*** "Warnings are ignored"
```

Note: The -export flag is important. Otherwise, the objects would not be accessible from outside Customer.dll.

Compiling and linking the stubs

First, you need to compile the stubs and the valuetype implementation to the object files. The output is:

```
*** "Generating the Customer_merged object"
   cl /I%OE_HOME%\corba\client\vb2ejb /I%OE_HOME%\corba\tools /I%MSVCDir%\include
/I%VBROKERDIR%\include /I%VBROKERDIR%\include\stubs /DWIN32 /D_USE_NAMESPACE /nologo /MD
/GX /c /DORB_VisiBroker -DSOM_DLL_Customer Customer_merged_c.cpp
```

```
Customer_merged_c.cpp
Customer_merged_c.hh(24) : warning C4067: unexpected tokens following preprocessor
directive - expected a newline
Customer_merged_c.hh(27) : warning C4067: unexpected tokens following preprocessor
directive - expected a newline

*** "Generating the value type interface objects"
   cl /I%OE_HOME%\corba\client\vb2ejb /I%OE_HOME%\corba\tools /I%MSVCDir%\include
/I%VBROKERDIR%\include /I%VBROKERDIR%\include\stubs /DWIN32 /D_USE_NAMESPACE /nologo /MD
/GX /c /DORB_VisiBroker -DSOM_DLL_Customer Vtlib_c.cpp
Vtlib_c.cpp

*** "Generating the value type implementation objects"
   cl /I%OE_HOME%\corba\client\vb2ejb /I%OE_HOME%\corba\tools /I%MSVCDir%\include
/I%VBROKERDIR%\include /I%VBROKERDIR%\include\stubs /DWIN32 /D_USE_NAMESPACE /nologo /MD
/GX /c /DORB_VisiBroker -DSOM_DLL_Customer Vtlib_I_vb.cpp
Vtlib_I_vb.cpp
```

Then you must link the stub library and DLL, which produces the output:

```
*** "Creating the Customer.lib library"
   lib /NOLOGO /LIBPATH:%OE_HOME%\corba\client\vb2ejb /LIBPATH:%OE_HOME%\corba\tools
/LIBPATH:%VBROKERDIR%\lib /OUT:Customer.lib /def: Customer_merged_c.obj Vtlib_c.obj
Vtlib_I_vb.obj
   Creating library Customer.lib and object Customer.exp

*** "Linking the Customer.dll library"
   link /DEBUG /NOLOGO /INCREMENTAL:NO /LIBPATH:%OE_HOME%\corba\client\vb2ejb
/LIBPATH:%OE_HOME%\corba\tools /LIBPATH:%VBROKERDIR%\lib /DLL Customer.exp
Customer_merged_c.obj Vtlib_c.obj Vtlib_I_vb.obj /OUT:Customer.dll
```

Building the tools library

The tools library is created from the same source files as in “Building the tools library” on page 167. But some parts of the implementation differ. This time we used different compiler flags, libraries, and include files. The creation of the tools library is done with a separate makefile, called `makefile.tools`. It is called this time from `makefile.vb2ejb`. The `makefile.tools` file executes the following steps:

1. Compile the tools classes. The following output is produced:

```
      set INCLUDES=/I%OE_HOME%\corba\client\vb2ejb /I%OE_HOME%\corba\tools
   /I%MSVCDir%\include  /I%VBROKERDIR%\include /I%VBROKERDIR%\include\stubs
      set CC_FLAGS=/DWIN32 /D_USE_NAMESPACE /nologo /MD /GX /c /DORB_VisiBroker
      set LIBS=/LIBPATH:%OE_HOME%\corba\client\vb2ejb /LIBPATH:%OE_HOME%\corba\tools
   /LIBPATH:C:\Inprise\vbroker\lib
      set LD_FLAGS=/DEBUG /NOLOGO /INCREMENTAL:NO /LIBPATH:%OE_HOME%\corba\client\vb2ejb
   /LIBPATH:%OE_HOME%\corba\tools /LIBPATH:%VBROKERDIR%\lib
      set LIB_FLAGS=/NOLOGO
      cd %OE_HOME%\corba\tools
      nmake -nologo -f makefile.tools

   *** "Compiling the Trace class"
      cl /I%OE_HOME%\corba\client\vb2ejb /I%OE_HOME%\corba\tools /I%MSVCDir%\include
   /I%VBROKERDIR%\include /I%VBROKERDIR%\include\stubs /DWIN32 /D_USE_NAMESPACE /nologo /MD
   /GX /c /DORB_VisiBroker -DSOM_DLL_tools Trace.cpp
   Trace.cpp

   *** "Compiling the Error class"
      cl /I%OE_HOME%\corba\client\vb2ejb /I%OE_HOME%\corba\tools /IC%MSVCDir%\include
   /I%VBROKERDIR%\include /I%VBROKERDIR%\include\stubs /DWIN32 /D_USE_NAMESPACE /nologo /MD
   /GX /c /DORB_VisiBroker -DSOM_DLL_tools Error.cpp
```

```
Error.cpp

*** "Compiling the Warning class"
   cl /I%OE_HOME%\corba\client\vb2ejb /I%OE_HOME%\corba\tools /I%MSVCDir%\include
/I%VBROKERDIR%\include /I%VBROKERDIR%\include\stubs /DWIN32 /D_USE_NAMESPACE /nologo /MD
/GX /c /DORB_VisiBroker -DSOM_DLL_tools Warning.cpp
Warning.cpp

*** "Compiling the MyOrb class"
   cl /I%OE_HOME%\corba\client\vb2ejb /I%OE_HOME%\corba\tools /I%MSVCDir%\include
/I%VBROKERDIR%\include /I%VBROKERDIR%\include\stubs /DWIN32 /D_USE_NAMESPACE /nologo /MD
/GX /c /DORB_VisiBroker -DSOM_DLL_tools MyOrb.cpp
MyOrb.cpp

*** "Compiling the CustomerWrapper class"
   cl /I%OE_HOME%\corba\client\vb2ejb /I%OE_HOME%\corba\tools /I%MSVCDir%\include
/I%VBROKERDIR%\include /I%VBROKERDIR%\include\stubs /DWIN32 /D_USE_NAMESPACE /nologo /MD
/GX /c /DORB_VisiBroker CustomerWrapper.cpp
CustomerWrapper.cpp
%OE_HOME%\corba\client\vb2ejb\Customer_merged_c.hh(24) : warning C4067: unexpected
tokens following preprocessor directive - expected a newline
%OE_HOME%\corba\client\vb2ejb\Customer_merged_c.hh(27) : warning C4067: unexpected
tokens following preprocessor directive - expected a newline

*** "Compiling the CustomerFactory class"
   cl /IF%OE_HOME%\corba\client\vb2ejb /I%OE_HOME%\corba\tools /I%MSVCDir%\include
/I%VBROKERDIR%\include /I%VBROKERDIR%\include\stubs /DWIN32 /D_USE_NAMESPACE /nologo /MD
/GX /c /DORB_VisiBroker CustomerFactory.cpp
CustomerFactory.cpp
%OE_HOME%\corba\client\vb2ejb\Customer_merged_c.hh(24) : warning C4067: unexpected
tokens following preprocessor directive - expected a newline
%OE_HOME%\corba\client\vb2ejb\Customer_merged_c.hh(27) : warning C4067: unexpected
tokens following preprocessor directive - expected a newline
```

Note: The usage of the flag /DORB_VisiBroker causes in some files different implementations to be chosen, for example in `MyOrb.cpp`. In these files, it is checked with `#ifdef` for `ORB_WebSphere` and `ORB_VisiBroker`.

2. Link the tools library and DLL, which produces the output:

```
*** "Creating the tools lib"
   lib /NOLOGO /LIBPATH:%OE_HOME%\corba\client\vb2ejb /LIBPATH:%OE_HOME%\corba\tools
/LIBPATH:%VBROKERDIR%\lib /OUT:tools.lib /def: Trace.obj Error.obj Warning.obj MyOrb.obj
CustomerWrapper.obj CustomerFactory.obj
   Creating library tools.lib and object tools.exp

*** "Linking the tools.dll library"
   link /DEBUG /NOLOGO /INCREMENTAL:NO /LIBPATH:%OE_HOME%\corba\client\vb2ejb
/LIBPATH:%OE_HOME%\corba\tools /LIBPATH:%VBROKERDIR%\lib Customer.lib /DLL tools.exp
Trace.obj Error.obj Warning.obj MyOrb.obj CustomerWrapper.obj CustomerFactory.obj
/OUT:tools.dll
   cd %OE_HOME%\corba\client\vb2ejb
   move %OE_HOME%\corba\tools\tools.dll .
```

Building the main object and linking the executable

The main class is `vb2ejb.cpp`. Beside the tools and stubs libraries, it uses a helper class to read the program arguments. The makefile executes the following steps:

1. Compile the main object. The following output is generated:

```
*** "Generating the vb2ejb.obj object"
   cl /I%OE_HOME%\corba\client\vb2ejb /I%OE_HOME%\corba\tools /I%MSVCDir%\include
/I%VBRODERDIR%\include /I%VBRODERDIR%\include\stubs /DWIN32 /D_USE_NAMESPACE /nologo /MD
/GX /c /DORB_VisiBroker vb2ejb.cpp
vb2ejb.cpp
Customer_merged_c.hh(24) : warning C4067: unexpected tokens following preprocessor
directive - expected a newline
Customer_merged_c.hh(27) : warning C4067: unexpected tokens following preprocessor
directive - expected a newline

*** "Compiling the ProgVals class"
   cl /I%OE_HOME%\corba\client\vb2ejb /I%OE_HOME%\corba\tools /I%MSVCDir%\include
/I%VBRODERDIR%\include /I%VBRODERDIR%\include\stubs /DWIN32 /D_USE_NAMESPACE /nologo /MD
/GX /c /DORB_VisiBroker ProgVals.cpp
ProgVals.cpp
```

2. Link the executable, which produces the output:

```
*** "Linking the vb2ejb.exe executable"
   link /DEBUG /NOLOGO /INCREMENTAL:NO /LIBPATH:%OE_HOME%\corba\client\vb2ejb
/LIBPATH:%OE_HOME%\corba\tools /LIBPATH:%VBRODERDIR%\lib vb2ejb.obj ProgVals.obj
Customer.lib %OE_HOME%\corba\tools\tools.lib  /SUBSYSTEM:CONSOLE /OUT:vb2ejb.exe
   Creating library vb2ejb.lib and object vb2ejb.exp
```

The output of the whole process is the executable vb2ejb.exe file.

Building the IOR tool

The IBM client uses a technique called *bootstrapping*. This allows this client to get the root context of the naming service.

For the VisiBroker client, this does not work. Therefore, we create a small tool that writes the string version of the root naming context's reference to a file. This file is then read by the client to connect to the naming service. The `makefile.vb2ejb` file creates the IOR tool and produces the following output:

```
*** "Creating the tool to get the naming root IOR"
   %WAS_HOME%\java\bin\javac com\ibm\itso\roch\wasaejb\NamingClient.java
```

Running the client

Before you can start the client, you have to create the file that stores the IOR of the root naming context. This is done by calling the following two commands:

```
cd %OE_HOME%\corba\vb2ejb
create_root_ior_file.bat
```

After that, you can call vb2ejb.exe. The general usage of the executable is:

```
vb2ejb method=<method> [id=<id>] [forename=<fname>] [lastname=<lname>] [phone=<phone>]
```

The following methods are available:

```
create [id=<id>] [forename=<fname>] [lastname=<lname>] [phone=<phone>]
```

Note: The vb2ejb client should be able to support the same functionality as the IBM client. But we have only implemented the *create()* method.

To create a customer with default values, you call:

```
vb2ejb method=create id=1234 forename=joe lastname=miller phone=01234-56789
```

The output of the executable would be like:

```
TRACE: Entering main ...
TRACE: Reading program arguments ...
TRACE: Program values:
TRACE: forename=joe, id=1234, lastname=miller, method=create, phone=01234-56789,
TRACE: Initializing the ORB ...
TRACE: Reading the Root Naming ContextIOR from nameservice.ior
TRACE: Getting name server ...
TRACE: By resolving the stringified IOR ...
TRACE: Constructing name entry ...
TRACE: Resolving name entry domain/legacyRoot/CustomerGlobal ...
TRACE: Narrowing CustomerHome ...
TRACE: Creating a CustomerBean with the following parameters:
TRACE: CustomerID: 1234
TRACE: ForeName:   joe
TRACE: LastName:   miller
TRACE: Phone:      01234-56789
TRACE: Customer created
TRACE: ( 1234, joe             , miller          , 01234-56789     )
```

The following subsections discuss in detail what happens when the vb2ejb.exe is executed:

- Class overview
- Initializing the VisiBroker ORB
- Getting the root context by resolving the stringified IOR
- Binding the CustomerHome
- Creating a customer

Class overview

The involved classes are the same as for the IBM client. Figure 8-7 on page 170 shows the class diagram.

You find the source code of the described classes in the directories:

```
%OE_HOME%\corba\client\vb2ejb
%OE_HOME%\corba\client\vb2ejb\com\ibm\itso\roch\wasaejb
%OE_HOME%\corba\tools
```

Initializing the VisiBroker ORB

After the *main* method reads the program parameters, the ORB is initialized. This is done inside the constructor of the `MyOrb` class. Example 8-5 shows the code.

Example 8-5 Initializing the VisiBroker ORB

```
MyOrb::MyOrb(string iorFileName)
{
   initOrb();
   _rootIOR = readIOR(iorFileName, "Root Naming Context");
}

void MyOrb::initOrb()
{
...
   int int0 = 0;

   //Initialize the ORB
   try {
      t.println("Initializing the ORB ...");
      _orb = CORBA::ORB_init(int0, NULL);
```

```
   }
...
}
```

Note that the initialization is slightly different from the one for the IBM client in Example 8-1 on page 171.

In addition, the string version of the naming services' root IOR has to be read from the file, as shown in Example 8-6.

Example 8-6 Reading the stringified root IOR

```
string MyOrb::readIOR(string fileName, string what)
{
   Trace t;
   Error err;
   char ior[10240];

   t.println("Reading the " + what + "IOR from " + fileName);
   ifstream in( fileName.c_str() );
   if (!in)
      err.println("Can not open File " + fileName);
   else {
      in >> ior;
      in.close();
   }

   return ior;
}
```

Getting the root context by resolving the stringified IOR

The next step is to get the root context of the naming service. Example 8-7 shows the method that handles this task. First the initial reference of the NameService is resolved by using *string_to_object()* and then narrowed to a `NamingContext` object.

Example 8-7 Get root context by resolving the stringified IOR

```
::CosNaming::NamingContext_ptr MyOrb::getRootContext()
{
...
   ::CosNaming::NamingContext_ptr rootContext = NULL;

   try {
      t.println("Getting name server ...");
      ::CORBA::Object_var obj;
...
      } else {
         t.println("By resolving the stringified IOR ...");
         obj = _orb->string_to_object( _rootIOR.c_str() );
      }
      if (CORBA::is_nil(obj)) {
         err.println("resolve_initial_references(\"NameService\") returned nil\nMake sure
your server is running and you set SOMCBPROPS!");
      }
      rootContext = ::CosNaming::NamingContext::_narrow(obj);
      if (CORBA::is_nil(rootContext)) {
```

```
            err.println("name server narrowed to nil");
         }
...
   return rootContext;
}
```

Again this step differs from the step for the IBM client shown in Example 8-2 on page 171.

Binding CustomerHome

The connection to the `CustomerHome` is established by the `CustomerFactory`. This step is exactly the same as described for the IBM client in Example 8-3 on page 172.

Creating a customer

After a `CustomerHome` is instantiated, all methods of the `CustomerHome` can be used. This section cover the creation of a customer. The other calls are similar.

The actual creation of the `Customer` is exactly the same like described for the IBM client in Example 8-4 on page 173. The only difference is the implementation of the helper method to convert between `WStringValue` and `char*`. For VisiBroker, we cannot use the method from the IBM `VtlUtil` class. We implement a simple version of that method. Example 8-8 shows this helper method.

Example 8-8 Conversion to WStringValue

```
::CORBA::WStringValue_ptr MyOrb::stringToWStringValue(char* str)
{
   try {
      int n = strlen(str)  + 1;
      ::CORBA::WChar *buffer = new ::CORBA::WChar[n];
      mbstowcs(buffer, str, n);
      return new ::CORBA::WStringValue(buffer);
   } catch(...) {
      return NULL;
   }
}
```

8.3.4 Problem determination

One major problem is how the clients find their servers. Therefore, it is good to check the entries in the namespace. Furthermore, it is a great help to get additional, CORBA communication traces output.

Checking the namespace

Two major mistakes concerning the namespace are:

- The server entry is not there at all (because the server may not have been started).
- The server entry has a different name than the client expects.

It is easy to find these mistakes if you simply print the complete namespace and check it for the entry your client is expecting. WebSphere Application Server is shipped with a tool called *dumpnamespace.bat* that gives you this functionality. You call:

```
setupcmdline
dumpnamespace
```

If it is not working, you should check if the administration server is running.

Enabling CORBA traces

It is possible to enable CORBA traces on the client and the server side to see what data is really sent by the client or received by the server.

IBM client

You have to set the following environment variables:

```
set SOMDCOMMDEBUGFLAG=259
set SOMDGETENV=1
```

You can simply delete the comment before those entries in the %OE_HOME%\corba\client\ibm2ejb\set_ibm2ejb_env.bat file and execute it on the command line.

It is best to redirect the output during execution of the client into a file because you get a large amount of trace output.

VisiBroker client

Unfortunately it looks like there are no possibilities to set environment variables or properties to get trace output from a VisiBroker client. It is possible for the VisiBroker daemons and for a server process, but not for a client.

WebSphere Administration Server

To get CORBA traces from the Administration Server, you have to edit the file %WAS_HOME%\bin\admin.config. You have to add the following configuration parameters:

```
com.ibm.CORBA.Debug=true
com.ibm.CORBA.CommTrace=true
com.ibm.ejs.sm.adminServer.traceString=ORBRas=all=enabled
com.ibm.ejs.sm.adminServer.traceOutput=%WAS_HOME%/logs/adminserver_orb.log
```

You can choose any location for the trace output file.

WebSphere default server

In the WebSphere Administrative Console, you have to change the JVM settings of the application server that runs the OrderEntry application (for example OrderEntryServer). You have to add the system properties as shown in Figure 8-9.

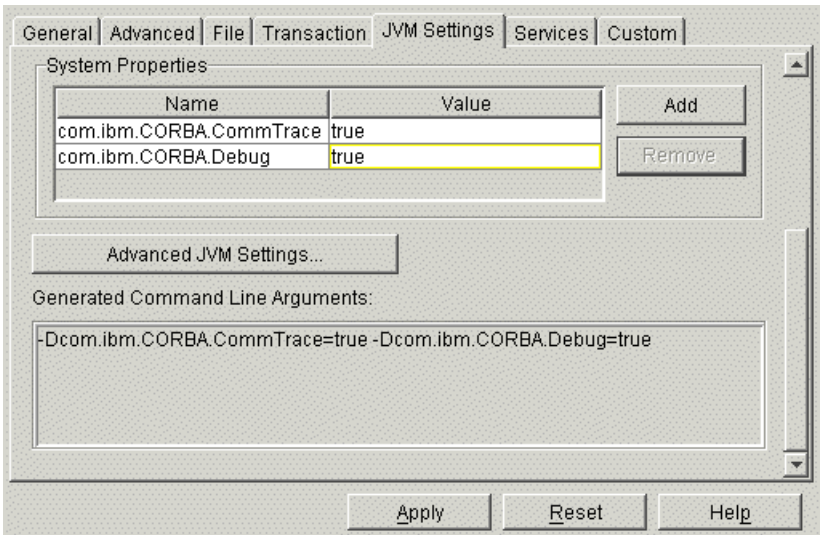


Figure 8-9 JVM settings for CORBA traces

These system properties activate the tracing for the Java ORB runtime.

Also, you have to set the trace filter and the trace location by using the WebSphere Application Server trace service. You achieve this by following these steps:

1. Click the **Services** tab (see Figure 8-9).
2. Select the **Trace Services** line.
3. Click **Edit properties**.
4. For the Trace specification value, add `SystemOut=all=disabled:ORBRas=all=enabled`.
5. Select the **Specify** radio button in the Trace output file field.
6. For the Trace output file (or another output file) value, add:

   `%WAS_HOME%/logs/orb_defaultserver.log`
7. Click **OK**.
8. Click **Apply**.
9. Restart your application server.

Figure 8-10 shows the settings for the trace specification.

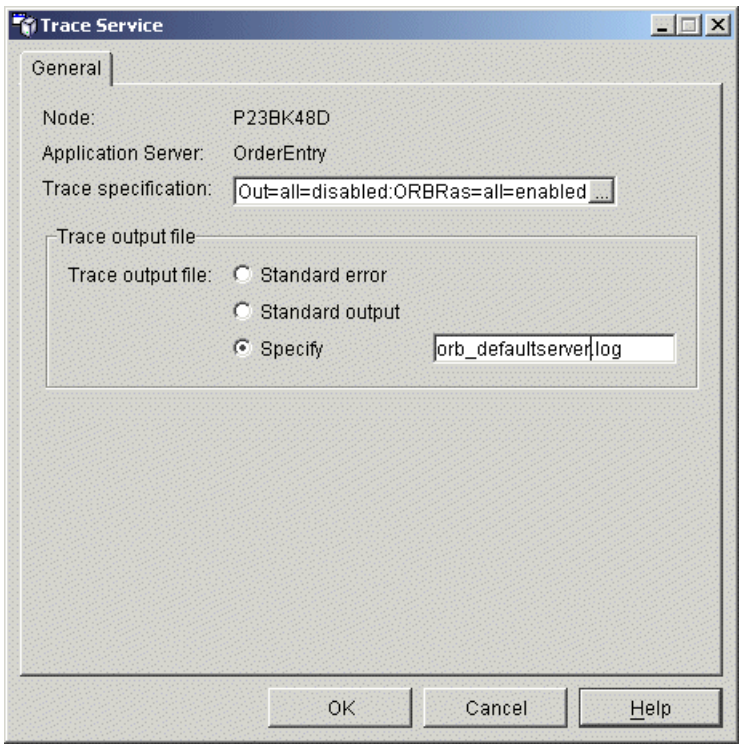


Figure 8-10 Filter and file settings for the CORBA traces

Note: IBM Distributed Debugger can be used as an additional problem determination tool. It enables source level debugging for both C++ and Java code.

8.4 Connecting OrderEntry application to C++ CORBA server

This section covers the opposite direction of interoperating with an EJB. In this scenario, the OrderEntry application is in the role of the client and not of the server.

We connect to a C++ server that offers the interface *CreditInstitute*. This interface can check the creditability of an OrderEntry customer. We only cover Orbix for the server implementation.

8.4.1 Configuring the environment

We used the following environment for our sample server:

- Windows 2000, SP 1
- WebSphere Application Server V4.0.1 Enterprise Edition
- Microsoft Visual C++ 6.0
- Orbix 2000 Enterprise Edition v2.0 Developers' Kit for C++ and Java

The installation and configuration of Windows, WebSphere Application Server, and Microsoft Visual C++ are not covered in this redbook. These are explained in the product specific installation guides.

Prerequisites

This scenario requires the following general prerequisites:

1. Extract the ZIP file containing the CORBA sample code to a directory.
2. Set the environment variable OE_HOME to the directory, where the ZIP file has been extracted.
3. Make sure that Microsoft Visual C++ is installed in the default directory C:\Program Files\Microsoft Visual Studio\VC98. If it is not, you have to edit the environment file %OE_HOME%\corba\set_common_env.bat.

Configuring Orbix

For this example, we used a 30-day evaluation version of Orbix 2000 V2.0. The following setup is described based on this experience. The details might slightly differ if you use a full version of the product.

All the online documentation, such as the installation, administrator's, and developer's guides for Orbix 2000, is available at: `http://www.iona.com/docs/orbix2000/2.0/index.html`

To install Orbix 2000, you have to follow these steps:

1. Register at: `http://www.iona.com/forms/orbix2000_eval_form.htm`
2. Download the software:
   a. Download the development package for Windows 2000.
   b. An additional window shows a link from where you can download the runtime package.
3. Install the packages like those that are described in the *IONA Orbix 2000 Installation Guide* (the chosen directory is referred to as <orbix_home>).
4. License the software:
   a. After registering, you receive mail with the license key as an attachment. The easiest way of licensing is to detach the file to the directory <orbix_home>\etc.
   b. Rename the file to `licenses.txt`.
5. Set the environment variable *JAVA_HOME* to the value of `%WAS_HOME%\java`.

Note: Microsoft Visual C++ 6.0 is the only C++ compiler that is officially supported by Orbix 2000 V2.0 (see the prerequisites in the *IONA Orbix 2000 Installation Guide*).

You have to configure Orbix 2000 after the installation. The configuration concept of Orbix 2000 is described in the *IONA Orbix 2000 Administrator's Guide, C++ Edition*, in the chapters “Selecting a Configuration Model” and “Configuring Orbix 2000”, which is available at:
http://www.iona.com/docs/orbix2000/2.0/admin_cpp/html/index.html

For the OrderEntry application, we use a simple local and no distributed configuration. However, you can create several configurations and activate them, if necessary.

To create a configuration domain in Orbix 2000, you should follow these steps:

1. Start the visual configuration tool by entering the following commands in a command prompt:

   ```
   cd <orbix_home>/orbix_art/2.0/bin
   itconfigure
   ```

2. Select **Create a configuration domain** from the pop-up window.
3. In the next screen, complete these tasks:
   a. Enter a name for the domain like **OrderEntry**.
   b. Select the **Standard services** radio button.
   c. Select the **Include configuration for demo applications** check box.
4. Click **Next**.
5. Select **Local File**, and click **Next**.
6. Leave the host properties as they are, and click **Next**.
7. Leave the infrastructure services as they are, and click **Next**.
8. Activate the check box for the **Naming Service**. Click **Next**.
9. Select the **Start all services via a startup script** radio button.
10. Click **Finish**.
11. Click **Deploy**.

After successful deployment, you see a display like the example in Figure 8-11. It tells you about scripts to be used for setting up the environment in a shell and for starting and stopping your services.

Orbix 2000 Configure
IONA
IONA Orbix 2000™
Configuration
Congratulations! Your configuration successfully deployed.
To set your environment to use this configuration domain, run the following:
C:\Program Files\iona\etc\bin\OrderEntry_env.bat
To start or stop the services associated with this domain, run the following:
C:\Program Files\iona\etc\bin\start_OrderEntry_services.bat
C:\Program Files\iona\etc\bin\stop_OrderEntry_services.bat
Close

Figure 8-11 Orbix configuration scripts

To use the created configuration batch files, you have to add the directory <orbix_home>\etc\bin to the PATH environment variable.

There are not many visual tools around with the Orbix installation, so you basically use the command prompt. To setup all environment variables properly, you would call:

```
OrderEntry_env.bat
```

In addition to these scripts, there is a command-line administration tool called **itadmin**. This tool is described in Chapter 16, “Managing Orbix Services with itadmin” in *IONA Orbix 2000 Administrator's Guide, C++ Edition*.

On Windows NT and 2000, you have the possibility to register your domain’s Orbix services as Windows services, so that you can start the services via the services panel. How this is done is described in Appendix A, “Orbix Windows NT Services” of *IONA Orbix 2000 Administrator's Guide, C++ Edition*.

Note: Several files are created during the Orbix 2000 configuration process. These files are called from the batch files supplied in the OrderEntry.zip file. See the last window in the configuration wizard (as shown in Figure 8-11) for the exact location of the generated files.

8.4.2 Connecting to an Orbix server

The following sections discuss what we had to do to create an Orbix server and connect the OrderEntry application to this server. We go through the basic steps, which are discussed in the following sections:

1. Set up the environment.
2. Build tools.
3. Define the IDL.
4. Build the server.
5. Run the server.
6. Build the test client.
7. Run the test client.
8. Build the client stubs for the EJB.
9. Run the EJB client.

You can find details about C++ and Java programming with Orbix 2000 V2.0 at the following sites:

- http://www.iona.com/docs/orbix2000/2.0/pguide_cpp/html/index.html
- http://www.iona.com/docs/orbix2000/2.0/pguide_java/html/index.html

Setting up the environment

To set up the necessary environment variables for the build steps and for running the server and the test client, you have to start a command prompt and execute:

```
%OE_HOME%\corba\server\ejb2orbix\set_ejb2orbix_env.bat
```

`OrderResourceBundle.properties` in the OrderEntry.ear file has to be edited as well. You have to set the following property (replace %OE_HOME% with the real path value):

```
nameservice.ior.directory=%OE_HOME%/corba/server/ejb2orbix
```

Before you can start the server, you have to start Orbix services (locator service, node daemon, and naming service) by entering:

```
start_OrderEntry_services
```

Building tools

Beside from the common build tools like **nmake**, C++ compiler, and so on, we use two other tools: **ant** and Orbix **genies**.

Ant

Ant is like Tomcat, a subproject of the Jakarta project from the Apache Software Foundation. Ant is a Java-based build tool that uses XML files as input. It is a more general form of **make**.

The Ant home page is at: http://jakarta.apache.org/ant/index.html

You can find the Ant user manual at: http://jakarta.apache.org/ant/manual/index.html

Orbix genies

Orbix 2000 has a feature called code generation toolkit. It is for the generation of a complete, working client and server directly from your IDL interfaces. The two basic components are the generator itself, `idlgen`, and several input TCL scripts, called `genies`. You get the available genies with the following call:

```
idlgen -list
```

It produces the output:

```
available genies are...
cpp_poa_equal.tcl    cpp_poa_random.tcl   java_poa_random.tcl
cpp_poa_genie.tcl    idl2html.tcl         list.tcl
cpp_poa_op.tcl       java_poa_genie.tcl   stats.tcl
cpp_poa_print.tcl    java_poa_print.tcl   vb_genie.tcl
```

We are using cpp_poa_genie.tcl for creating our CORBA server and java_poa_genie.tcl for creating a Java test client.

You can also use the toolkit to generate an HTML documentation of your IDLs. This is done with the command:

```
idlgen idl2html.tcl CreditInstitute.idl
```

You find more about the code generation toolkit for Orbix 2000 V2.0 in the document: http://www.iona.com/docs/orbix2000/2.0/codegen/html/index.html

Defining the IDL

The starting point for the server development is to create the IDL file. It shall represent an `CreditInstitute` object that offers the possibility to check the creditability of a customer. The only method is *isCreditableFor()*, which takes a `Person` object and a double as input parameters. It returns true or false and can throw two kinds of the `CORBA::UserExceptions` exceptions. See Example 8-9 for the details.

Example 8-9 The CreditInstitute interface

```
module com {
module ibm {
module itso {
module roch {
module credit {

   const string ROOT_IOR_FILENAME = "nameservice.ior";
   const string OE_NAMING_DIR = "OrderEntry";
   const string CREDIT_NAMING_ENTRY = "CreditInstitute";
```

```
   struct Person {
      string forename;
      string lastname;
   };

   exception PersonNotFound {string why; };
   exception InformationConfidential { string why; };

   interface CreditInstitute {
      boolean isCreditableFor(
         in Person pers,
         in double pay
      ) raises (PersonNotFound, InformationConfidential);
   };

}; // credit
}; // roch
}; // itso
}; // ibm
}; // com
```

The nested module structure is mapped to the package name. There are three constants that define the name of the root IOR file and the entry in the naming service. The constants will be used from the clients and the server.

Building the server

To build the server, you simply have to run a makefile by executing the following two commands:

```
cd %OE_HOME%\corba\server\ejb2orbix
nmake /f makefile.ejb2orbix server
```

This file executes all the steps that are described in the following subsections:

1. Generate the server sources
2. Modify the generated server sources
3. Compile the server code

We show the output of the commands that are executed by the makefile. You don't have to execute any of the commands manually. But you could do so if you are interested in building the client manually step by step.

The **nmake** may execute the commands in a different order because it doesn't follow a sequence of commands, but resolve dependencies. The **nmake** command is described in detail in the online documentation of Microsoft Visual C++.

The build of the server takes place in the subdirectory "srv".

To delete all the generated files and directories, you call the two commands:

```
cd %OE_HOME%\corba\server\ejb2orbix
nmake /f makefile.ejb2orbix cleansrv
```

Generating the server sources

The makefile calls first the Orbix Code Generation Toolset in order to create the sources and the makefile for the C++ server. It not only creates the server main, but also delivers an implementation for the interface, which sends back random answers. `makefile.ejb2orbix` generates the server sources, which produces the following output:

```
   idlgen cpp_poa_genie.tcl -ns -all -dir srv CreditInstitute.idl
CreditInstitute.idl:
cpp_poa_genie.tcl: creating srv/it_servant_base_overrides.h
cpp_poa_genie.tcl: creating srv/it_servant_base_overrides.cxx
cpp_poa_genie.tcl: creating srv/com_ibm_itso_roch_credit_CreditInstituteImpl.h
cpp_poa_genie.tcl: creating srv/com_ibm_itso_roch_credit_CreditInstituteImpl.cxx
cpp_poa_genie.tcl: creating srv/server.cxx
cpp_poa_genie.tcl: creating srv/client.cxx
cpp_poa_genie.tcl: creating srv/call_funcs.h
cpp_poa_genie.tcl: creating srv/call_funcs.cxx
cpp_poa_genie.tcl: creating srv/it_print_funcs.h
cpp_poa_genie.tcl: creating srv/it_print_funcs.cxx
cpp_poa_genie.tcl: creating srv/it_random_funcs.h
cpp_poa_genie.tcl: creating srv/it_random_funcs.cxx
cpp_poa_genie.tcl: creating srv/Makefile
```

Note: As you can see, the genie creates a client as well. However, there have been compilation problems with that client. For that reason, we create additionally the Java test client as described in “Building the test client” on page 200.

Modifying the generated server sources

You can compile and run the generated sources without any changes. We did only some minor changes to the server main and the interface implementation.

The makefile copies over the two edited files. The output is shown here:

```
*** "Copy over the completed server files"
   copy /y srv\completed\server.cxx srv
        1 file(s) copied.
   copy /y srv\completed\com_ibm_itso_roch_credit_CreditInstituteImpl.cxx srv
        1 file(s) copied.
```

In the files that are copied over, we changed:

- In `server.cxx`:
  - The name of the server entry in the namespace
  - The root IOR of the namespace is written to a file
- In `com_ibm_itso_roch_credit_CreditInstituteImpl.cxx`:

  In the algorithm for the random answers, we reduced the frequency, how often the two exceptions are thrown and added, and more meaningful exception text.

Compiling the server code

`idlgen` creates a makefile that is used for the compilation of the server code:

- The IDL compilation produces the following output:

```
   *** "Compiling the server sources"
      cd srv
      nmake -nologo -f Makefile server.exe

      "<orbix_home>\orbix_art\2.0\bin\idl" -base -poa  -I"<orbix_home>\orbix_art\2.0\idl"
   F:/Henning/oe/corba/server/ejb2orbix/CreditInstitute.idl
```

- The output from the creation of the object files is:

```
      if exist server.obj del server.obj
      cl -c -I"<orbix_home>\orbix_art\2.0\cxx\include"   -Zi -nologo -GR -GX -W3 -Zm200
   -MD        -Foserver.obj server.cxx
   server.cxx
      if exist CreditInstituteC.obj del CreditInstituteC.obj
```

```
   cl -c -I"<orbix_home>\orbix_art\2.0\cxx\include"   -Zi -nologo -GR -GX -W3 -Zm200
-MD        -FoCreditInstituteC.obj CreditInstituteC.cxx
CreditInstituteC.cxx
   if exist it_random_funcs.obj del it_random_funcs.obj
   cl -c -I"<orbix_home>\orbix_art\2.0\cxx\include"   -Zi -nologo -GR -GX -W3 -Zm200
-MD        -Foit_random_funcs.obj it_random_funcs.cxx
it_random_funcs.cxx
   if exist it_print_funcs.obj del it_print_funcs.obj
   cl -c -I"<orbix_home>\orbix_art\2.0\cxx\include"   -Zi -nologo -GR -GX -W3 -Zm200
-MD        -Foit_print_funcs.obj it_print_funcs.cxx
it_print_funcs.cxx
   if exist CreditInstituteS.obj del CreditInstituteS.obj
   cl -c -I"<orbix_home>\orbix_art\2.0\cxx\include"   -Zi -nologo -GR -GX -W3 -Zm200
-MD        -FoCreditInstituteS.obj CreditInstituteS.cxx
CreditInstituteS.cxx
   if exist com_ibm_itso_roch_credit_CreditInstituteImpl.obj del
com_ibm_itso_roch_credit_CreditInstituteImpl.obj
   cl -c -I"<orbix_home>\orbix_art\2.0\cxx\include"   -Zi -nologo -GR -GX -W3 -Zm200
-MD        -Focom_ibm_itso_roch_credit_CreditInstituteImpl.obj
com_ibm_itso_roch_credit_CreditInstituteImpl.cxx
com_ibm_itso_roch_credit_CreditInstituteImpl.cxx
   if exist it_servant_base_overrides.obj del it_servant_base_overrides.obj
   cl -c -I"<orbix_home>\orbix_art\2.0\cxx\include"   -Zi -nologo -GR -GX -W3 -Zm200
-MD        -Foit_servant_base_overrides.obj it_servant_base_overrides.cxx
it_servant_base_overrides.cxx
```

- The output from linking the executable is:

```
   link /out:server.exe /DEBUG /NOLOGO  server.obj  CreditInstituteC.obj
it_random_funcs.obj  it_print_funcs.obj  CreditInstituteS.obj
com_ibm_itso_roch_credit_CreditInstituteImpl.obj  it_servant_base_overrides.obj
/LIBPATH:"<orbix_home>\orbix_art\2.0\cxx\lib"   it_naming.lib   it_genie.lib
kernel32.lib ws2_32.lib advapi32.lib user32.lib it_dynany.lib it_poa.lib  it_art.lib
it_ifc.lib
   cd F:\Henning\oe\corba\server\ejb2orbix
```

Running the server

You run the server by executing two commands:

```
cd %OE_HOME%\corba\server\ejb2orbix
start_server
```

The output of the executable is like:

```
Initializing the ORB
Writing IOR of root naming context to file nameservice.ior...
Waiting for requests...
```

Note: If you get an error, make sure that the Orbix services are started by executing the **`start_OrderEntry_services`** command.

The following subsections explain in detail what is happening when server.exe is started:

- Class overview
- Initializing the Orbix ORB
- Creating a POA
- Creating and activating servant objects
- Connecting to nameservice
- Exporting object reference
- Putting the ORB into an active state
- Handling requests

Class overview

The main program handles basically the `POA_Manager`, the `POA`, and a `CreditInstitute` objects. The `CreditInstitute` object is the C++ interface class that is derived from the IDL description like the `InformationConfidential` and `PersonNotFound` exception classes. The implementation for *`CreditInstitute`* interface is done by `CreditInstitute_Impl`. It is derived from the Orbix specific class `IT_Servant_BaseOverrides`, which itself is derived from the general CORBA class `ServantBase`. The `CreditInstitute_Impl` class uses two more generated helper classes: `IT_GenieRandom` and `IT_GeniePrint`. The first is one for generating random answers, and the second one is for printing trace information.

Figure 8-12 shows the class diagram for these classes.

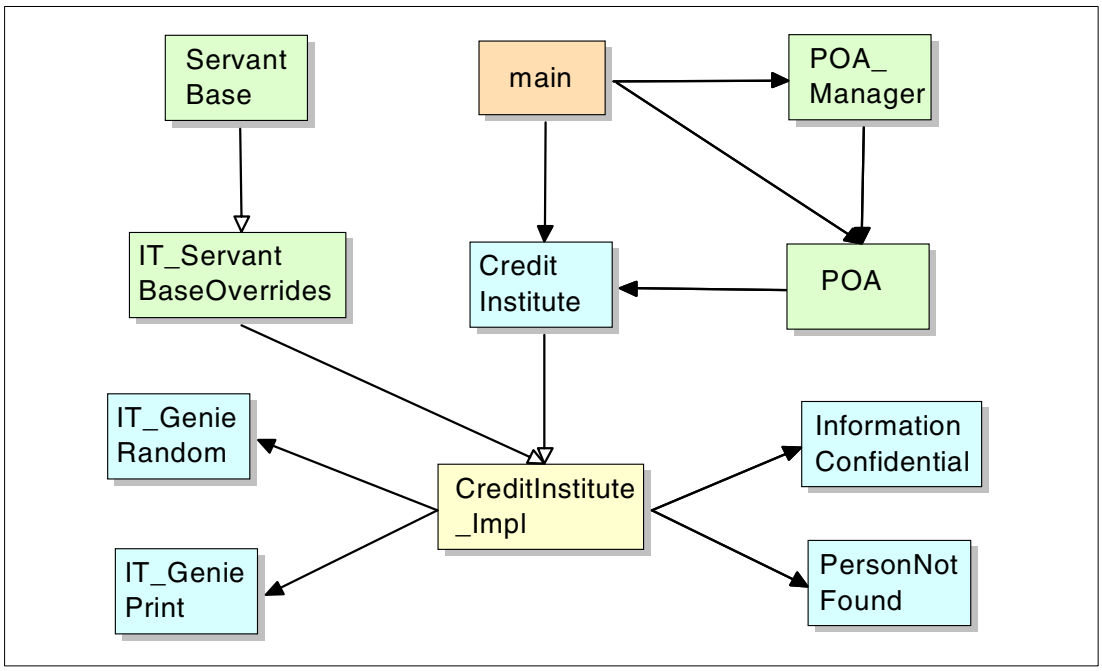


Figure 8-12 Class diagram for ejb2orbix

You can divide the classes into three groups:

- CORBA or Orbix base classes:
  - `POA_Manager`
  - `POA`
  - `ServantBase`
  - `IT_ServantBaseOverrides`
- IDL generated classes:
  - `CreditInstitute`
  - `InformationConfidential`
  - `PersonNotFound`
  - `IT_GenieRandom`
  - `IT_GeniePrint`
- Manually created classes:
  - `main`
  - `CreditInstitute_Impl`

Initializing the Orbix ORB

The first step is to initialize the ORB. Example 8-10 shows the code fragments.

Example 8-10 Initializing the Orbix ORB

```
CORBA::ORB_var global_orb = CORBA::ORB::_nil();
...
CORBA::Object_var tmp_ref;  // For temporary object references.
...
cout << "Initializing the ORB" << endl;
global_orb = CORBA::ORB_init(argc, argv);
...
```

Creating a portable object adapter (POA)

The object adapter is the link between the ORB and the servants (implementation objects of the interfaces). The POA provides the server-side runtime support that connects server application code to the networking layer of the ORB.

Example 8-11 shows how a POA is created.

Example 8-11 Creating a POA

```
PortableServer::POA_ptr create_simple_poa(
    const char*                   poa_name,
    PortableServer::POA_ptr       parent_poa,
    PortableServer::POAManager_ptr poa_manager )
{
    // Create a policy list. Policies not set in the list get default values.
    //
    CORBA::PolicyList policies;
    policies.length(1);
    int i = 0;
    // Make the POA single threaded.
    //
    policies[i++] = parent_poa->create_thread_policy(PortableServer::SINGLE_THREAD_MODEL);
    assert(i==1);

    return parent_poa->create_POA(poa_name, poa_manager, policies);
}
...
tmp_ref = global_orb->resolve_initial_references("RootPOA");
PortableServer::POA_var root_poa = PortableServer::POA::_narrow(tmp_ref);
assert(!CORBA::is_nil(root_poa));
PortableServer::POAManager_var root_poa_manager = root_poa->the_POAManager();
assert(!CORBA::is_nil(root_poa_manager));
...
PortableServer::POA_var my_poa = create_simple_poa("my_poa", root_poa, root_poa_manager);
```

First you get access to the root POA object by bootstrapping. Next, you obtain a reference to the root `POA manager` object, which manages a set of `POA`s. Then you instantiate your own `POA`, which defines several policies. These policies determine the thread handling, how objects are activated, and so on. We define a single-threaded `POA` for transient objects.

Creating and activating servant objects

Next, the implementation object is instantiated and activated. Activation means that the object is registered with the `POA`. `POA` returns a unique identifier. Example 8-12 shows how this is done in the generated code.

Example 8-12 Create and activate servant object

```
PortableServer::Servant the_com_ibm_itso_roch_credit_CreditInstitute = 0;
PortableServer::ObjectId_var oid;
...
the_com_ibm_itso_roch_credit_CreditInstitute =
com_ibm_itso_roch_credit_CreditInstituteImpl::_create(my_poa);
oid = my_poa->activate_object(the_com_ibm_itso_roch_credit_CreditInstitute);
```

Connecting to nameservice

The next steps are shown in Example 8-13. The server connects to the Orbix nameservice by bootstrapping and writes the IOR of the root context into a file.

Example 8-13 Connecting to nameservice and writing IOR

```
void writeIOR (CORBA::Object_ptr obj, const char *fileName)
{
   char iorfile[100];

   iorfile[sizeof(iorfile)-1] = 0;
   strncpy(iorfile, fileName, sizeof(iorfile)-1);

   std::ofstream outFile;
   outFile.open(iorfile);
   if (outFile.fail())
      std::cerr << "Cannot open file " << iorfile << std::endl;
   else {
      CORBA::String_var string_ref = global_orb->object_to_string(obj);

      outFile << (char *)(string_ref) << std::endl;
      outFile.close();

      std::cout << "Writing IOR of root naming context to file " << fileName << "..." <<
std::endl;
   }
}
...
tmp_ref = global_orb->resolve_initial_references("NameService");
CosNaming::NamingContextExt_var default_context =
CosNaming::NamingContextExt::_narrow(tmp_ref);
assert(!CORBA::is_nil(default_context));

writeIOR(default_context, com::ibm::itso::roch::credit::ROOT_IOR_FILENAME);
```

Exporting object reference

The next step is to export the object reference to the namespace. Since we don't want to export our reference directly under the naming root, we first have to create a naming context. Underneath that, we put the `CreditInstitute` reference. The code fragments are shown in Example 8-14.

Example 8-14 Exporting object reference

```
CosNaming::NamingContext_ptr create_demo_context(
    CosNaming::NamingContextExt_ptr default_context,
    const char* demo_name_string ) IT_THROW_DECL (( CORBA::Exception ))
{
    CORBA::Object_var context_as_obj;

    // First see if the name is already bound.
    //
    try
    {
        context_as_obj = default_context->resolve_str(demo_name_string);
    }
    catch (const CosNaming::NamingContext::NotFound&)
    {
        // Name is not yet bound, create a new context.
        //
        CosNaming::Name_var demo_name =
            default_context->to_name(demo_name_string);
        context_as_obj = default_context->bind_new_context(demo_name);
    }

    // Narrow the reference to a naming context.
    //
CosNaming::NamingContext_var demo_context =
        CosNaming::NamingContext::_narrow(context_as_obj);
    if (CORBA::is_nil(demo_context)) {
        cout << "Error: Cannot narrow naming context called `"
             << demo_name_string <<  "'." << endl;
        throw CORBA::BAD_PARAM();
    }
    return demo_context._retn();
}

CosNaming::NamingContext_var demo_context = create_demo_context(default_context,
com::ibm::itso::roch::credit::OE_NAMING_DIR);
...
tmp_ref = my_poa->id_to_reference(oid);
name = default_context->to_name(com::ibm::itso::roch::credit::CREDIT_NAMING_ENTRY);
demo_context->rebind(name, tmp_ref);
```

Putting the ORB into an active state

The last step for starting up the server is to activate the `POA manager` and to instruct the `ORB` to listen for client requests. This is shown in Example 8-15.

Example 8-15 Letting the server listen for requests

```
root_poa_manager->activate();
cout << "Waiting for requests..." << endl;
global_orb->run();
```

Now the server is ready to handle requests.

Handling requests

The implementation of the *CreditInstitute* interface is done by the class com_ibm_itso_roch_credit_CreditInstituteImpl. If a method request comes in to the ORB, it is forwarded to that class. Example 8-16 shows how the *isCreditableFor()* method is implemented.

Example 8-16 Handling requests

```
CORBA::Boolean com_ibm_itso_roch_credit_CreditInstituteImpl::isCreditableFor(
    const com::ibm::itso::roch::credit::Person& pers,
    CORBA::Double pay ) IT_THROW_DECL(( CORBA::SystemException,
    com::ibm::itso::roch::credit::PersonNotFound,
    com::ibm::itso::roch::credit::InformationConfidential ))
{
    cout << "com_ibm_itso_roch_credit_CreditInstituteImpl::isCreditableFor(): called." <<
endl;
    CORBA::Boolean                  _result;
...
//
    _result = global_random->get_boolean();

    //
    // Decide if we want to throw back an exception.
    //
    switch (global_random->get_range(10)) {
    case 1:
        {
            com::ibm::itso::roch::credit::PersonNotFound IT_ex("No person with the given
lastname found");
...
            throw IT_ex;
        }
    case 2:
        {
           com::ibm::itso::roch::credit::InformationConfidential IT_ex("Even the reason is
confidential");
...
            throw IT_ex;
        }
    default:
        // Don't throw an exception.
        //
        break;
    }
...
    return _result;
}
```

The global_random class is a helper class that is created by genie for creating random values. *get_boolean()* delivers a true value with a probability of 50%. An exception is thrown with a probability of 20%.

If you send 10 requests to this method, you will get on average:

- True (four times)
- False (four times)
- PersonNotFound (1 time)
- InformationConfidential (1 time)

Building the test client

The test client gives you the possibility to test the server without starting WebSphere Application Server and running the OrderEntry application. To build the test client, you simply have to run a makefile:

```
cd %OE_HOME%\corba\server\ejb2orbix
nmake /f makefile.ejb2orbix testclt
```

This execute all the steps, which are described in the following subsections:

- Generating the test client sources
- Modifying the generated test client sources
- Compiling the test client code

The build of the test client takes place in the subdirectory “testclt”. To delete all the generated files and directories, you call:

```
cd %OE_HOME%\corba\server\ejb2orbix
nmake /f makefile.ejb2orbix cleantestclt
```

Generating the test client sources

The Orbix Code Generation Toolset is used to create the sources and the build file for the Java test client. The output of the makefile execution is shown here:

```
*** "Generating the test client sources"
   idlgen java_poa_genie.tcl -ns -all -dir testclt CreditInstitute.idl
CreditInstitute.idl:
java_poa_genie.tcl: creating testclt/idlgen/IT_GenieRandom.java
java_poa_genie.tcl: creating
testclt/TestPackage/com/ibm/itso/roch/credit/CreditInstituteCaller.java
java_poa_genie.tcl: creating testclt/TestPackage/client.java
java_poa_genie.tcl: creating
testclt/TestPackage/com/ibm/itso/roch/credit/CreditInstituteImpl.java
java_poa_genie.tcl: creating testclt/TestPackage/server.java
java_poa_genie.tcl: creating testclt/build.xml
Type "ant" to compile the generated code
```

Note: If you are experiencing problems, check if your CLASSPATH environment variable name is defined all in uppercase. That is the way the TCL script expects it to be set. For Windows, it doesn’t make a difference, but for genie it does.

Modifying the generated test client source

You can compile and run the generated sources without any changes. We only changed the name of the server entry in the namespace. The makefile copies over the one edited file. The output is:

```
*** "Copy over the completed test client files"
   copy /y testclt_completed\client.java testclt\TestPackage
        1 file(s) copied.
```

Compiling the test client code

`idlgen` creates the ant type build file build.xml. This file is used for the compilation of the test client code. The steps for compiling test client code are:

1. The IDL compilation produces the following output:

```
*** "Compiling the test client sources"
   cd testclt
   ant -buildfile build.xml
Buildfile: build.xml
```

```
init:
    [mkdir] Created dir: F:\Henning\oe\corba\server\ejb2orbix\testclt\classes

idl_compile:
```

2. From the Java compile, you get the following output:

```
build_all:
Invalid argument
Invalid argument
Invalid argument
There are no more files
There are no more files
There are no more files
There are no more files
Invalid argument
Invalid argument
    [javac] Compiling 24 source files to
F:\Henning\oe\corba\server\ejb2orbix\testclt\classes

BUILD SUCCESSFUL
```

The warnings can be ignored.

Running the test client

You run the test client by executing two commands:

```
cd %OE_HOME%\corba\server\ejb2orbix
start_testclient
```

The test client produces no output. But in the server window, you would see the following output:

```
Waiting for requests...
com_ibm_itso_roch_credit_CreditInstituteImpl::isCreditableFor(): called.
    pers = struct Person {
        forename = "xy"
        lastname = ""
    }
    pay = 9.75
com_ibm_itso_roch_credit_CreditInstituteImpl::isCreditableFor(): returning...
    _result = false

com_ibm_itso_roch_credit_CreditInstituteImpl::isCreditableFor(): called.
    pers = struct Person {
        forename = "xy"
        lastname = ""
    }
    pay = 9.75
com_ibm_itso_roch_credit_CreditInstituteImpl::isCreditableFor(): returning...
    _result = true
```

Building the client stubs for the EJB

To connect the OrderEntry application to the CreditInstitute server, the interface had to be compiled with the WebSphere Application Server IDL compiler.

To build the client stubs, you have to execute two commands:

```
cd %OE_HOME%\corba\server\ejb2orbix
nmake /f makefile.ejb2orbix ejb
```

To delete all the generated files and directories, you can call:

```
cd %OE_HOME%\corba\server\ejb2orbix
nmake /f makefile.ejb2orbix cleanejb
```

The makefile calls `ant` in order to complete the build:

1. The IDL compile produces the following output:

```
*** "Compiling the ejb client stubs"
   ant -buildfile ejb\build_client_stubs.xml
Buildfile: ejb\build_client_stubs.xml

build_idl:
     [echo] Compiling IDL sources
     [exec] Parsing ..\CreditInstitute.idl
     [exec] done  - ..\CreditInstitute.idl
     [exec]
     [exec] Generating com
     [exec] done   -   com
```

2. From the compilation of the Java classes, we only show fragments of the output:

```
build_classes:
     [echo] Compiling java sources
     [exec] [parsing started com\ibm\itso\roch\credit\_CreditInstituteImplBase.java]
     [exec] [parsing completed 271ms]
     [exec] [parsing started com\ibm\itso\roch\credit\_CreditInstituteStub.java]
     [exec] [parsing completed 180ms]
...
     [exec] [loading
C:\IBM\WebSphere\AppServer\java\jre\lib\rt.jar(org/omg/CORBA_2_3/portable/ObjectImpl.cla
ss)]
     [exec] [loading
C:\IBM\WebSphere\AppServer\java\jre\lib\rt.jar(org/omg/CORBA/portable/ObjectImpl.class)]
...
     [exec] [checking com.ibm.itso.roch.credit.PersonNotFoundHolder]
     [exec] [wrote com\ibm\itso\roch\credit\PersonNotFoundHolder.class]
     [exec] [checking com.ibm.itso.roch.credit.ROOT_IOR_FILENAME]
     [exec] [wrote com\ibm\itso\roch\credit\ROOT_IOR_FILENAME.class]
     [exec] [total 8913ms]
```

3. The classes are packaged into a JAR file. The output is:

```
build_jar:
      [jar] Building jar: F:\Henning\oe\corba\server\ejb2orbix\ejb\CreditInstitute.jar

BUILD SUCCESSFUL

Total time: 26 seconds
```

This compilation doesn't have to be done because the generated classes are already packaged inside the EAR file. It is only needed if the interface was changed.

Running the EJB client

Before an order is placed, `ApplicationManager` checks the customer's credit history. The result is placed in the `ViewBean` object. Figure 8-13 shows the result in case an exception is thrown.

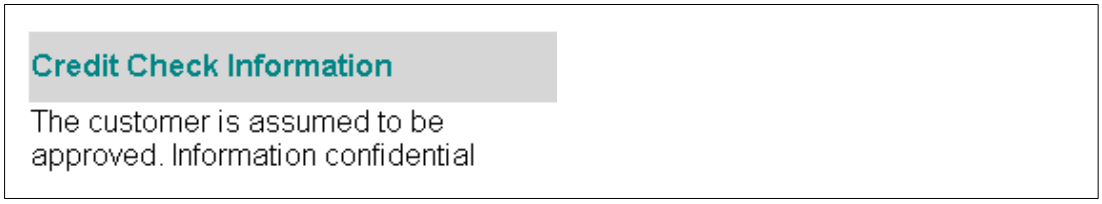


Figure 8-13 Results of creditability check

The server shows the answer as well in its output:

```
com_ibm_itso_roch_credit_CreditInstituteImpl::isCreditableFor(): called.
    pers = struct Person {
        forename = "xy"
        lastname = ""
    }
    pay = 9.75
com_ibm_itso_roch_credit_CreditInstituteImpl::isCreditableFor(): throwing exception
InformationConfidential {
    why = "Even the reason is confidential"
}
```

The following subsections explain what is happening when the creditability is checked:

- Class overview
- Initializing the Java client ORB
- Getting the root context by resolving the stringified IOR
- Binding the CreditInstitute object

Class overview

The `ApplicationManager` class accesses the `CreditInstituteWrapper` class, which hides the complexity of CORBA processing. Example 8-17 shows the code for performing a credit history check. The *isCreditableFor()* method is called with the customer's first name, last name, and the total of the order as parameters.

Example 8-17 Calling the CreditInstituteWrapper class

```
private CreditInstituteWrapper institute = null;

public boolean checkCreditability()
{
   boolean isApproved = true;
   String info = "The customer is assumed to be approved. ";

   // check connection
   if (this.institute == null) {
      try {
         this.institute = new CreditInstituteWrapper();
      }
...
   }

   // connection is available
   try {
      Support.Customer customer = itemsViewBean.getCustomer();
      double orderTotal = itemsViewBean.getOrderTotal();
      isApproved = this.institute.checkCreditability(
                        customer.getFirstName(),
```

```
                        customer.getLastName(),
                        orderTotal);
      info = "";
   } catch (PersonNotFound ex) {
      // no error situation
      info += "Customer not found in CreditInstitute";
      System.out.println(info);
      ex.printStackTrace();
   } catch (InformationConfidential ex) {
      // no error situation
      info += "Information confidential";
      System.out.println(info);
      ex.printStackTrace();
   }

   itemsViewBean.setIsApproved(isApproved);
   itemsViewBean.setApprovedInfo(info);
   return isApproved;
}
```

Since the wrapper forwards the exceptions, the `ApplicationManager` has the possibility to decide how to react on the user exceptions. If a user exception occurs, the customer is assumed to be creditable.

Initializing the Java client ORB

The constructor of the `CreditInstituteWrapper` reads the name of the file that stores the server IOR (see Example 8-13 on page 197) from the OrderResourceBundle.properties file and passes it to the `ClientOrb` constructor.

The initialization of the Java client `ORB` is similar to the ones for the C++ clients shown in “Initializing the IBM ORB” on page 171 and “Initializing the VisiBroker ORB” on page 183. Example 8-18 shows the listing of the `CreditInstituteWrapper` class constructor.

Example 8-18 Initializing the Java client ORB

```
   public CreditInstituteWrapper()
   {
      ResourceBundle resources = ResourceBundle.getBundle("OrderResourceBundle");
      String iorFile = resources.getString("nameservice.ior.directory");
      iorFile += System.getProperty("file.separator");
      iorFile += ROOT_IOR_FILENAME.value;

   String nameEntry = OE_NAMING_DIR.value + "/" + CREDIT_NAMING_ENTRY.value;
   org.omg.CORBA.Object instObj = null;
...
      ClientOrb orb = new ClientOrb(iorFile);
...
   }

public class ClientOrb {
...
   private ORB           orb = null;

   public ClientOrb(String filename) {
...
      Properties orbProps = new Properties();
      orbProps.put("org.omg.CORBA.ORBClass", "com.ibm.CORBA.iiop.ORB");
```

```
      this.orb = ORB.init((String[]) null, orbProps);
...
   }
```

In addition, the string version of the naming services' root IOR has to be read from the file. This process is shown in Example 8-19.

Example 8-19 Reading the Orbix naming IOR

```
   private String        rootIOR = null;
...
         rootIOR = readIOR(filename, "Root Naming Context");

   private String readIOR(String fileName, String what)
                         throws FileNotFoundException, IOException
   {
      System.out.println("reading " + what + " IOR file: " + fileName);
      BufferedReader in = new BufferedReader(new FileReader(fileName));
      String ior = in.readLine();

      return ior;
   }
```

Getting root context by resolving the stringified IOR

Example 8-20 shows how the root context of the Orbix naming service is resolved by invoking the *string_to_object()* method in the `ORB` object.

Example 8-20 Getting the Orbix root naming context

```
   public NamingContext getRootContext() {
      NamingContext rootContext = null;
...
      org.omg.CORBA.Object nameServiceObject = null;

      System.out.println("Getting name server ...");
      System.out.println("By resolving the stringified IOR ...");
      nameServiceObject = this.orb.string_to_object(this.rootIOR);
      if (nameServiceObject!=null) {
         rootContext = NamingContextHelper.narrow(nameServiceObject);
      }

      // check success
      if (rootContext == null)
         throw new java.lang.RuntimeException("Unable to locate naming context");

      return rootContext;
   }


NamingContext rootContext = orb.getRootContext();
```

Binding to the CreditInstitute object

Binding to the `CreditInstitute` object is done as well in the constructor of the wrapper class as shown in Example 8-21.

Example 8-21 Binding the CreditInstitute object

```
         System.out.println("Constructing name entry ...");
         NameComponent[] nameArray = ClientOrb.stringToName(nameEntry);

         System.out.println("Resolving name entry " + nameEntry + " ...");
         instObj = rootContext.resolve(nameArray);

   public static NameComponent[] stringToName(String nameString)
   {
      System.out.println("Creating NameComponent for " + nameString);
      StringTokenizer tok = new StringTokenizer(nameString, "/");

      int partNo = tok.countTokens();

      NameComponent name[] = new NameComponent[partNo];
      for (int i=0; i<partNo; i++) {
         if (tok.hasMoreElements()) {
            String token = tok.nextToken();
            name[i] = new NameComponent(token, "");
         } else
            System.err.println("Token not found");
      }

      return name;
    }
```

After that, the wrapper is ready to be used by the `ApplicationManager` object.

8.4.3 Problem determination

The two major building blocks for problem determination are checking the namespace and the additional CORBA communication debug output.

Checking the namespace

As we mentioned in “Configuring Orbix” on page 188, the Orbix administration tool is `itadmin`. You can either use `itadmin` as command-line utility or as command shell. The general usage is:

```
itadmin <object> <method> <parameters>
```

The object that represents the namespace is `ns`. You can execute the following commands:

1. Get a listing of the namespace:

   ```
   itadmin ns list
   ```

2. Get a listing of the OrderEntry naming context:

   ```
   itadmin ns list OrderEntry
   ```

3. Resolve the binding to the `CreditInstititute` object:

   ```
   itadmin ns resolve OrderEntry/CreditInstitute
   ```

Enabling CORBA traces

Orbix 2000 has a logging subsystem. You get different levels of logging reports (for example warning, information, and so on) that can be filtered and redirected to files. Also, useful problem determination could be accomplished by enabling the tracing for WebSphere EJB Java client.

The logging output is separated based on the configuration domain. For the OrderEntry domain, you get several files written to the <orbix_home>\var\OrderEntry\logs directory. The default settings include logging for the locator, naming, and the node daemon service.

You can edit the logging properties for the OrderEntry domain in the file <orbix_home>\etc\domains\OrderEntry.cfg.

You can find more details about the Orbix 2000 logging subsystem in *IONA Orbix 2000 Administrator's Guide, C++ Edition* in Chapter 14, “Managing Orbix Logging”, at:
http://www.iona.com/docs/orbix2000/2.0/admin_cpp/html/index.html

9

The Business Process Beans service

This chapter discusses the Business Process Beans (BPBeans) service, including some technical details about the service. Since the BPBeans service is a technical preview and have several limitations, we do not use the service in the OrderEntry application. Rather, we discuss the design and development steps in creating such a solution.

The following topics are discussed:

- BPBeans basics
- BPBeans programming model
- BPBeans development
- Potential usage of the BPBeans service in the OrderEntry application

Note: To understand the structure of the OrderEntry application and to run it, read the instructions in Appendix A, “Downloading and installing the OrderEntry application” on page 221.

© Copyright IBM Corp. 2002

9.1 BPBeans basics

BPBeans framework is a new technology that provides the advanced functionality to extend the J2EE programming model. There are three main directions in which the BPBeans service expands the J2EE model:

- It supports long running (or extended) business transactions. These long running business transactions include multiple J2EE transactions. This approach allows a more flexible and more sophisticated Enterprise Application Integration (EAI) solutions to be developed than would be possible by using the J2EE model alone.
- It provides a compensation mechanism. The J2EE model operates only on the traditional *ACID transactions*. ACID transactions use a simple model where the resources can be locked for the duration of a transaction. This requires the transactions to be small and fast to avoid holding those resource locks for significant periods of time.

  The advantage of having a long-running transaction or many transactions managed as a single unit of work (UOW) requires (in case of process failure) a new mechanism to "roll-back" or compensate for changes that have already been committed. The compensation mechanism doesn't dictate an "all or nothing" rule for the process. Rather it invokes a second process to analyze the results of the failed process and perform the steps to compensate for the failure.

  For example, think of a process that runs multiple J2EE transactions in a single UOW. Failure of one of the J2EE transactions doesn't mean that the committed transactions have to be rolled back. It is more intelligent to re-run the failed transaction (or perform some other steps) and continue with the process.
- It allows the building of complex solutions from existing Java components using a visual, rather than coding, environment. The solution is built using a hierarchy of different BPBeans components. These components are then connected and grouped together to form larger, reusable components. A special modeling tool, called Application Builder for Components (ABC), is used to build a hierarchy of different BPBeans components.

The whole idea of the BPBeans framework is to make the application building process easier, faster, and more productive by:

- Supporting the advanced business process modeling mechanism
- Promoting reusability of BPBeans components as much as possible
- Separating control logic from the Java components to the easily-modifiable model files (stored in XMI format)
- Providing a non-programmatic way of changing the application behavior by modifying the BPBeans properties
- Providing a server environment for testing of an application model.

To put it another way, the BPBeans components and the server in which they reside represent a workflow system.

A typical application built upon the BPBeans framework looks like the diagram shown in Figure 9-1.

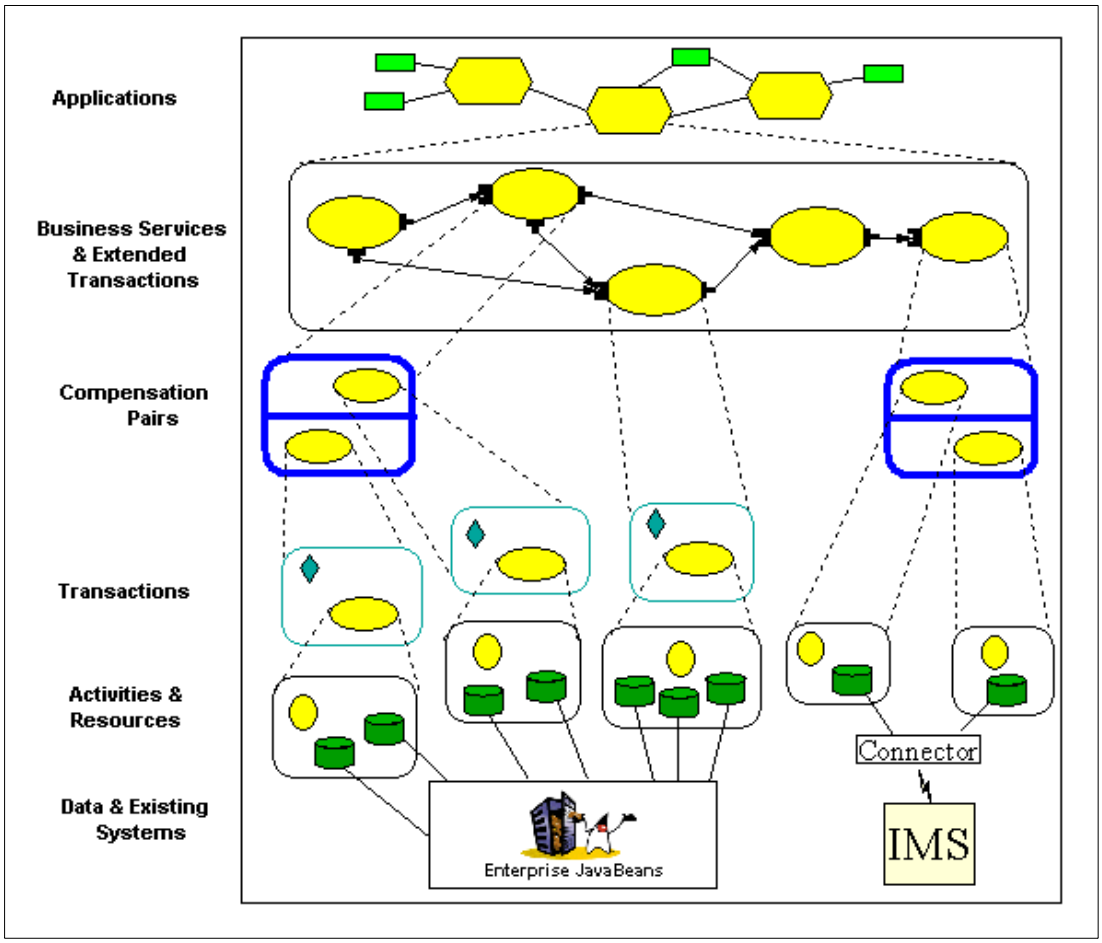


Figure 9-1 Typical application built with BPBeans technology

The interesting thing about this diagram is that only the lower layers in this model contain the Java components that need to be coded. All other layers and objects in this model are pre-built. You just assemble them together in a certain way and customize their behavior by changing the component's properties in the ABC tool.

An application is built using the "divide and conquer" idea of assembling a complex application from smaller and simpler components. Thus, for example, *activities* and *resources* are built based on the Java components that are implemented by a Java developer: EJBs, servlets, JavaBeans, and so on. The activities and resources, in turn, are the building blocks for transaction processes and compensation pairs (you can find more on this in "Process" on page 213).

To support the highest level of reusability, all BPBeans components define their external interface in a declarative manner – through the component's properties. This declaration describes the interfaces of the components together with requirements on the environment in which the component may be used.

The components are connected to each other using messages. A component is given work via input messages and can generate work for other components via output messages. However a given component does not know who gave it the work or to whom its messages will be given. Such association between components is called a *loosely coupled connection*.

As a result, a component has almost no dependency on any other component and can be built separately from the rest of the application. Also, a BPBeans component can be assembled with any other BPBeans component that satisfy its external interface (as described in “Activity” on page 212).

However this reusability comes with a cost in the designing phase. All external interfaces for the BPBeans components have to be carefully designed.

9.2 BPBeans programming model

The BPBeans programming model is based on the variety of the BPBeans components. A BPBeans component is a JavaBean class that follows the rules of the BPBeans framework and can be plugged into this framework. Some components are included as part of the framework; others need to be created by a Java developer. Each component has a set of well-defined responsibilities in the BPBeans application.

We discuss the following components:

- Task:
  - Activity
  - Process
- Resource
- Context
- Outcome Decider

Note: For more information about the BPBeans components, refer to *Business Process Beans Programming Guide*, which is part of the online documentation.

9.2.1 Task

A piece of work in the BPBeans model that performs some business function is called a *task*. A task receives and sends messages through objects called *input* and *output ports*. There are two types of tasks in the BPBeans model:

- **Activity**: The simplest task that implements a piece of business logic
- **Process**: A task that manages a collection of other tasks (activities or other processes)

Activity

An activity is a BPBeans component that represent a basic task in the BPBeans model – the smallest piece of business logic. An activity has to be created by a Java developer. It must produce a result, called an *outcome*, in the BPBeans framework. All possible outcomes for a given activity are grouped in the object called *outcome set*. Depending on the logic of an activity, the outcome can be of different types: boolean or integer.

A JavaBean class has to implement the `com.ibm.bpbeans.Activity` interface to be treated as an activity in the BPBeans framework. The `com.ibm.bpbeans.Activity` interface declares just a few methods that an activity has to implement. It doesn’t impose any restrictions on the business methods. A Java developer determines the business methods a given activity should have.

Note: Actually a JavaBeans class that does *not* implement the *com.ibm.bpbeans.Activity* interface can be used as an activity, but the JavaBean cannot send messages to other activities. It can only receive messages (and, optionally, send a single reply to the message).

An activity is invoked by receiving a message on its input port.

Note: There are four types of ports in the BPBeans framework:

- Input port
- InOut port
- OutIn port
- Output port

Receiving a message results in the execution of a method. The method that should be invoked and the input parameter type are declared as the properties for this input port. For example, the `ReceiveActivity` activity (part of the BPBeans tutorials) has input and output ports. The input port has three properties that define how another BPBeans component can send a message (aka invoke) to this input port:

- The property `Java class name` defines the name of the Java class/interface that implements or declares the functionality of this port. In our example, the *com.ibm.bpbeans.InputPort* interface is declared. The `InputPort` interface has two methods: one is to add an input port listener and another is to remove an input port listener. Any component that implements the *com.ibm.bpbeans.InputPortListener* interface can register with an input port in order to receive a message. In simple cases, when there is only one BPBeans component on the input port, you can use a shortcut by assigning a value to the property called `Input Method`, which is described next.
- The property `Input Method` defines the name of the method that should be invoked when a message arrives to an input port. This property is used when you have *only one* component listening on the input port. In our example, the property is set to `setInputMessage`. This method is part of the activity class, `ReceiveActivity`, not the *InputPort* interface. Basically, this property provides a shortcut for routing an incoming message directly to the activity, bypassing the input port object.
- The input parameter type (aka Java class name of the input parameter) is declared as the `Data type` property, and it is set to `com.ibm.bpb.tutorials.MessagePassing.DateMessage`.

The BPBeans framework provides the `com.ibm.bpb.base.ActivityBase` class that implements the *com.ibm.bpbeans.Activity* interface. In most cases, you need to extend this class in order to create your new Activity. The `com.ibm.bpb.base.ActivityBase` class provides the implementation of the required methods for the `Activity` components. Plus it adds several utility methods that will be useful with almost any activity (for example, the *sendMessage()* method or the *getResource(java.lang.String resourceName)* method, and so on).

Process

Process in the BPBeans framework is responsible for managing all activities and other processes that it contains. A process defines:

- How it creates, starts, or stops an instance of an activity or process
- How the outcomes are handled
- How transactions are handled

To be as flexible as possible, the BPBeans framework provides several process types:

- **Activity wrapper process**: This process provides a way to combine Activity, Resource (described later), and transaction support (optional, but usually included as part of the process). Transaction support is achieved by adding a BPBeans component, called *Unit of Work (UOW) Director*, to the process. UOW Director demarcates the transaction boundary: All activities and processes contained inside Activity Wrapper Process are executed as a single transaction. UOW Director supports open nested transactions.
- **Compensation pair process**: This process type combines two tasks:
  - The first task is the main task and performs some business logic.
  - The second task is a compensation task. It is executed only when the main task fails. The steps that the second task executes depend on the application logic; they can be as simple as rolling back the changes, or they might be more complex.
- **Concurrent process**: This process type supports execution of the multiple tasks in parallel.
- **Instance cluster process**: This type of process provides a way to create a separate instance of a task for each value of the input parameter. For example, you can create an e-mail managing process, where all messages for a particular user are managed by a separate instance of an e-mail task. In other words, an instance cluster process checks the value of a user ID parameter in all messages. When it receives a message, it tries to find a task that is responsible for managing the e-mail messages for a given user. If such a task exists, the process routes an e-mail message to this task. If a task doesn't exist, the instance cluster process creates a new task.
- **Keep alive process**: This process performs two steps whenever a new message arrives on its input port:
  - Check if a running task exists. It it does, the process routes the incoming message to this task.
  - If there is no running task, the process creates a new task and routes the incoming message to this task.
- **Message cluster process**: This process creates a new task for each incoming message.
- **Repeat process**: This process runs contained tasks a specified number of times or until some condition is satisfied.
- **Retry process**: This process runs the contained tasks repeatedly either until it succeeds or it reaches the predefined threshold limit for the number of attempts.
- **Sequential process**: This process runs the contained tasks sequentially. The order of the execution is specified during modeling of the application in the ABC tool.
- **Source cluster process**: This process creates a separate task for sending each outgoing message.
- **Swap process**: This process represents a finite state machine; the process executes a specific task for each state. This process also contains a default task that is executed when the state is undefined.
- **Time limit process**: This process stops a contained instance of a task if it runs longer than the predefined time limit. This is achieved through a configurable property.

9.2.2 Resource

A Resource component represents a source of data (database, legacy application, and so on). In order for a JavaBeans class to be treated as a Resource component, it has to implement the `com.ibm.bpbeans.Resource` interface.

The BPBeans framework provides the `com.ibm.bpb.base.ResourceBase` class that implements the *`com.ibm.bpbeans.Resource`* interface. In most cases, you just need to extend this class to implement your Resource. The `com.ibm.bpb.base.ResourceBase` class implements the required methods for a `Resource` component, plus it provides several utility methods.

9.2.3 Context

Context acts like a mediator between a BPBeans components and the runtime environment (similar to a security context or transaction context in WebSphere Application Server). It provides the following services to a BPBeans component:

- Manages a component's life cycle
- Manages the context properties
- Monitors a component's status

The runtime environment starts and stops the BPBeans components through the context object. The context properties are name-value pairs. These properties are set either during the modeling of an application in the ABC tool or at the runtime by the BPBeans runtime.

The monitoring service provides the way to access the current state of a BPBeans component. It also provides the way to view and record the debugging information.

The BPBeans framework provides several specialized context components:

- Activity context
- Resource context
- Outcome decider context
- Process context
- JavaBean context

Any BPBean components have to be *wrapped* inside a corresponding context in order to participate in the BPBeans model. The word "wrapped" comes from the terminology used in the BPBeans framework. Another way to look at the same concept is to realize that each component has to have an association with a context object. As we discussed earlier, a context object provides some services to a BPBeans component. For example, Activity Context performs the following actions:

- Gets a port for Activity
- Publishes the outcome of Activity, so that it is available to other BPBeans components
- Gets a Resource component used by Activity
- Checks Activity's state

9.2.4 Outcome decider

A result of a process execution is defined through its outcome. In some cases, like in case of a Concurrent process, the outcome of a process is established by analyzing the outcomes from all contained processes in a special component – *Outcome decider*. This component can analyzes the outcomes of the *same* type.

In the case when tasks in a Concurrent process produce outcomes of different types, a BPBeans component, called *Outcome Mapper*, is used. Outcome Mapper converts the outcomes from one type to the outcomes of another type. This approach of using specialized objects to convert and analyze the result of a task helps to achieve the high reusability of the BPBeans components.

9.3 Generic steps in BPBeans development

Despite the variety of the BPBeans components, there are generic steps in developing an application in the BPBeans framework.

Important: This section should be used in conjunction with BPBeans Javadoc and the programming guide to completely understand and use the BPBeans framework.

The steps are outlined here:

1. Design the application. Since this step is big in itself, we don't concentrate on this step because it is not the topic of this section.
2. Develop the following BPBeans components in any Java IDE:
   - **Activity**: Activity has to implement the *com.ibm.bpbeans.Activity* interface (or it can extend the `com.ibm.bpb.base.ActivityBase` class). Depending on the type of port Activity uses, you need to implement a different interface in Activity:
     - ***Input Port***: Implement the *com.ibm.bpbeans.InputPortListener* interface.
     - ***InOut Port***: This port is designed to receive a message and send a response. Implement the *com.ibm.bpbeans.InOutPortListener* interface. When the message arrives, the *requestArrived* method is invoked. One of the parameters in this method is of type `com.ibm.bpbeans.ReplyToken`. This parameter is used to send the reply.
     - ***OutIn Port***: This port is opposite to InOut Port. First, a message is sent, then Activity waits for a response. It would be valid to assume that Activity should implement the *com.ibm.bpbeans.OutInPortListener* interface. However, to receive a response, Activity has to implement the *com.ibm.bpbeans.ResponseListener* interface. An example of using this type of port is shown in Example 9-1.
     - ***Output Port***: You use this port if Activity needs just to send the message without expecting a reply. You don't need to implement any interface in Activity unless Activity needs to know when the message was really sent. In this case, Activity should implement the *com.ibm.bpbeans.OutputPortListener* interface.

     Also, code all business logic for this Activity.
   - **Resource**: Resource has to implement the *com.ibm.bpbeans.Resource* interface (or it can extend the `com.ibm.bpb.base.ResourceBase` class). Code all business logic for accessing a data source.
   - **Outcome Decider**: If you plan to use Concurrent Process, you need to develop Outcome Decider. It has to implement the *com.ibm.bpbeans.OutcomeDecider* interface. By analogy with other components, the BPBeans framework has a class that implements the *com.ibm.bpbeans.OutcomeDecider* interface. As a result, it is easier to extend the `com.ibm.bpb.base.OutcomeDeciderBase` class.
   - **Outcome Set and Outcome Mapper**: The BPBeans framework provides numerous classes for Outcome Set and Outcome Mapper. In most cases, you use supplied classes. However, if you want to create a new component, follow this guideline:
     - Outcome Set has to implement the *com.ibm.bpbeans.OutcomeSet* interface. For ease of use, the BPBeans framework provides the implementation of two classes: `com.ibm.bpb.base.FiniteOutcomeSetBase` and `com.ibm.bpb.base.InfiniteOutcomeSetBase`. Use one of them for your Outcome Set.

- Outcome Mapper has to implement the *com.ibm.bpbeans.OutcomeMapper* interface. To facilitate the development process, the BPBeans framework supplies the com.ibm.bpb.base.OutcomeMapperBase class that should be extended.

3. Import all the components that you developed in the previous step into the ABC tool.
4. Add the description for all developed components.
5. Wrap Activities, Resources, and Outcome Decider into the corresponding context (Activity can be also wrapped in Activity Wrapper Process).
6. Repeat the following steps as many times as your application logic requires. Remember that you start assembling the application from the most basic components. Gradually, you create more complex components until you assemble the whole application (refer to Figure 9-1 on page 211).

   The steps include:

   a. Select the specific components, and wrap them in the appropriate process type.
   b. Make the necessary connections between the components in the process.
   c. Define the necessary context properties.
7. Add the Client Type to the BPBeans model. Client Type in the model represents the entry point for the components (not necessarily BPBeans components) outside the BPBeans application to invoke it.
8. Connect Client Type to the input ports of the BPBeans application.
9. Export the application model from the ABC tool. The tool creates a JAR file with the application model and a batch file to start the application.
10. Develop an application client code. This is the code from where you will invoke your BPBeans application. EJBs, JSPs, or servlets are good candidates for this step.

    Example 9-1 shows the sample code for invoking the BPBeans application. We used OrderEntryClerk EJB from the OrderEntry application to write this code.

Example 9-1 Sample code from the OrderEntryClerk EJB

```
...
// Begin Business Process Beans
import com.ibm.bpb.client.local.LocalClientContext;
import com.ibm.bpbeans.ResponseListener;
import com.ibm.bpbeans.ResponseToken;
import com.ibm.bpbeans.IncompatibleMessageException;
import com.ibm.bpbeans.OutInPort;

import com.ibm.itso.roch.bpb.message.PlaceOrderMessage;
import com.ibm.itso.roch.bpb.message.NoReplyMessage;
// End Business Process Beans

/**
 * This is a Session Bean Class
 */
public class OrderEntryClerkBean implements SessionBean,
                                            ResponseListener (1)
{
   // Begin Business Process Beans
    private Object _lock = new Object(); (2)
    private PlaceOrderMessage _answer = null;
   // End Business Process Beans

...........Some code is omitted here.........................
```

```
   // Following is the state in this stateful session bean

   public String custID;
   public Vector items;
   public Vector myCart;
   public DataSource ds;
   boolean isDirty;


// Begin Business Process Beans method support
  public String bpbPlaceOrder() throws IncompatibleMessageException
  {
   String applicationLocation =
"C:\\OrderEntry\\BusinessProcessBeans\\deployed\\OrderEntry\\"; (3)
   String application         = "OrderEntryApp";               (4)
   String clientType          = "PlaceOrderClient";            (5)

   LocalClientContext bpbClientCtx =
     new LocalClientContext(applicationLocation,application,clientType); (6)

   OutInPort oip = bpbClientCtx.getOutInPort("PlaceOrderMessage"); (7)

   PlaceOrderMessage oMsg =
     new PlaceOrderMessage(Integer.parseInt(custID),
                           Integer.parseInt(whid),
                           Integer.parseInt(did),
                           items);                          (8)
    ResponseToken token = oip.sendRequest(oMsg); (9)
    token.addResponseListener(this); (10)
    synchronized(_lock)
    {
      while (_answer == null)
      {
        System.out.println("OrderEntryClerkBean.pbpPlaceOrder() before locking until answer
arrives from BPBean Application.");
        try {  _lock.wait(); } // Block the current thread
        catch(InterruptedException e)
        {
        }
      } // End while
      System.out.println("OrderEntryClerkBean.pbpPlaceOrder() awoke because answer arrived
from BPBean Application.");
    }
    // _answer set by call to responseArrived before unlocking this thread.
    return Integer.toString(_answer.getOrderId());
  }

  // Called by BPBeans from a BPBeans thread
  public void allRespondersClosed(ResponseToken rt) (11)
  {
    synchronized(_lock)
    {
      if (_answer == null)
        _answer = new NoReplyMessage();
      _lock.notify();
    }
  }

  // Called by BPBeans from a BPBeans thread
```

```
  public void responseArrived(ResponseToken rt, java.io.Serializable orderId) (12)
  {
    synchronized(_lock)
    {
      if (_answer == null)
      {
        if (orderId instanceof PlaceOrderMessage)
          _answer = (PlaceOrderMessage) orderId;
        _lock.notify();
      }
    }
  }

// End Business Process Beans method support
...
```

The most interesting lines are highlighted. The following list explains each line using corresponding numbers in superscript:

(1) This class implements the *ResponseListener* interface. It means that this class uses OutIn Port; first it send a message, and second it waits for response. The *ResponseListener* interface declares two methods that are used by the BPBeans runtime to notify the listener when a response comes.

(2) One of the limitations in the BPBeans service in WebSphere Application Server Enterprise Edition Version 4.0 is that it doesn't have a synchronous mechanism to send and receive message between an application client and a BPBeans application. For this reason, we use a Java object to lock and wait for the response from the application. However, even this solution has a problem. In those environments where BPBeans can create threads (which might actually include the WebSphere Application Server EJB container), the call back is made by BPBeans on a thread created by BPBeans and, therefore, not on the thread blocked by the wait call. These limitations should be removed in the production version of the BPBeans service.

(3) This `String` represents the location of the application model file. The file is created by the ABC tool and will be in the repository subdirectory of the tool. The file will have the same name as the application, but with `.Application.xmi` appended to the end of the name. If you exported your application, this parameter should specify the location of a JAR file.

(4) This `String` defines the name of the BPBeans application as it is specified in the application model.

(5) This `String` defines the name of Client Type (created in the step 7. on page 217) as it is specified in the application model.

(6) To invoke a BPBeans application, you need to create the `com.ibm.bpb.client.local.LocalClientContext` object. You specify three parameters that identify the application. In the technology preview, a BPBeans application has to be located on the same system as your client code. The client code also directly accesses the file system. These limitations should be removed in the production version of the BPBeans service.

(7) By invoking this method on the `LocalClientContext` object, we request the OutIn Port object. It will be used to send a message to the application.

(8) We construct the message that will be sent to the application. In the BPBeans framework, we highly recommend that you create a message class either by implementing the *com.ibm.bpbeans.Message* (or *com.ibm.bpbeans.ClientMessage*) interface or by extending the `com.ibm.bpb.base.MessageBase` class.

(9) Send a message to the application. The method returns an object that implements the *com.ibm.bpbeans.ResponseToken* interface. This interface has a method that allows any object, that implements *ResponseListener* interface, to register with `ResponseToken`.

(10) Register the OrderEntryClerkBean object as `ResponseListener` with `ResponseToken`.

(11) This method is part of the *com.ibm.bpbeans.ResponseListener* interface.

(12) This method is part of the *com.ibm.bpbeans.ResponseListener* interface. It is invoked when a response for a message comes back.

Note the usage of the `_lock` object in Example 9-1 on page 217.

A

Downloading and installing the OrderEntry application

If you want to run the sample application or to browse the source code, you need to download the ZIP file from the Redbooks Web page at: http://www.redbooks.ibm.com

This appendix explains:

- Downloading the OrderEntry application ZIP file
- Unzipping the OrderEntry application ZIP file
- Configuring WebSphere Application Server
- Installing the OrderEntry application

© Copyright IBM Corp. 2002

Setup instructions

The following sections provide instructions for downloading and installing the OrderEntry application files. The assumption are:

- You are familiar with administrative tasks for WebSphere Application Server
- You have enough authority to access and manipulate DB2 and WebSphere Application Server

Environment

The development described in this redbook was done using the following configuration:

- **Operating system**: Windows 2000
- **Database**: IBM DB2 UDB v7.2
- **Application server**: WebSphere Application Server Enterprise Edition v4.0
- **Development environment**: WebSphere Studio Application Developer
- Make sure you have the WAS_HOME environment variable set; it should point to the installation directory for WebSphere Application Server (for example C:\IBM\WebSphere\AppServer)
- We use the %WAS_HOME% and %OE_HOME% (defined later in the appendix) environment variables throughout the book to specify the installation directories for WebSphere Application Server and the OrderEntry application respectively

All instructions throughout this book assume the environment described above. If you use a different environment, adjust the instructions in the book accordingly.

Downloading the OrderEntry application ZIP file

The OrderEntry application ZIP file contains all the setup files, source and compiled code. To download the file, you need to perform the following steps:

1. Open the IBM Redbooks Web page at: `http://www.redbooks.ibm.com/`
2. Type `SG24-6504` in the search window, and click **Search**.
3. Follow the link for this redbook.
4. In the Web page, click the **Additional Materials** link in the right part of the window.
5. Save the ZIP file on your system.

Unzipping the file

Follow these steps to unzip the OrderEntry ZIP file:

1. Open the OrderEntry ZIP file in WinZip or a similar program.
2. Unzip the OrderEntry ZIP file to any directory (for example: C:\OrderEntry).
3. Add the environment variable *OE_HOME*. Set the value of OE_HOME to the directory name to where you unzipped the file. For example, set it to `C:\OrderEntry`.

Description of the directory structure

When you unzip the OrderEntry ZIP file, you should see a directory structure under %OE_HOME%. There are two EAR files in the root directory:

- **OrderEntry.ear**: Contains all the source and compiled code for the OrderEntry application.
- **OrderEntryWithoutBRBeans.ear**: Is for those who want to go through the process of developing BRBeans. The process is described in Chapter 3, “The Business Rule Beans service” on page 15.

The %OE_HOME% directory also contains several subdirectories:

- **DB2**: This directory contains the script files necessary to create the database used by the OrderEntry application.
- **ActiveX, BRBeans,** and **corba**: These directories contain files specific to a corresponding WebSphere Enterprise Service.
- **WebSphereWorkbench**: This directory contains several files that you can use in WebSphere Studio Application Developer. This tool creates a special file called *.classpath* for each project in the workspace. The file contains the location of all projects, folders and JAR files that are referenced by the current project (refer to the WebSphere Studio Application Developer online help for the details on changing the class path). If you plan to use WebSphere Studio Application Developer, you can copy and paste the content of a file from the WebSphereWorkbench directory into a corresponding .classpath file in WebSphere Studio Application Developer. For example, copy the content of *%OE_HOME%\WebSphereWorkbench\BRBeans.classpath* and paste it into the *.classpath* file located in *BRBeans* project in WebSphere Studio Application Developer (see Figure A-1). This step sets up the class path for the projects with the references to all required directories and JAR files.

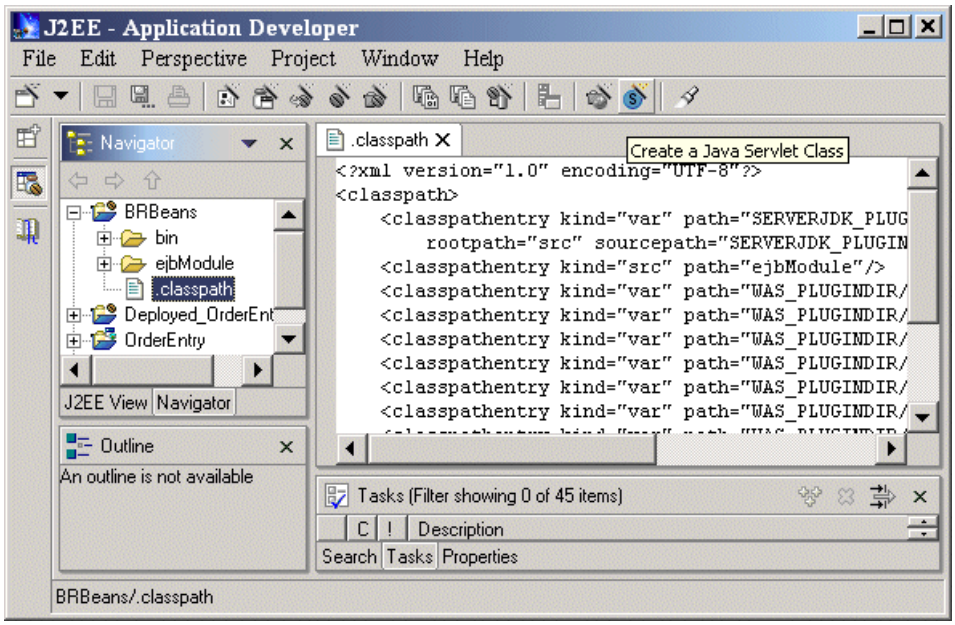


Figure A-1 Editing a .classpath file

Setting up the databases

You need to set up two databases:

- The OrderEntry application database
- The BRBeans database

Setting up the OrderEntry application database

The process of creating and populating the OrderEntry database is straightforward if you use Windows 2000 or Window NT:

1. Open a command prompt.
2. Change the directory to `%OE_HOME%\DB2`.
3. Run the following command:

   ```
   CreateDB_OrderEntry.bat
   ```

4. The CreateDB_OrderEntry.bat file logs setup information to the C:\temp\OrderEntry.log file. Examine this file. You need to fix any problems before you attempt to install and run the OrderEntry application.

Setting up the BRBeans database

To create a BRBeans database, follow these steps:

1. Open a command prompt.
2. Change the directory to `%OE_HOME%\BRBeans`.
3. Run the following command:

   ```
   brbdb.bat
   ```

4. The script logs the setup information to the C:\Temp\BRBeansSetup.log file. Examine this file. You need to fix any problem before you attempt to install and run the OrderEntry application.

Configuring WebSphere Application Server

Before you can install the OrderEntry application, you have to create two datasources (one to access the OrderEntry database and another the BRBeans database) and create an application server.

To create the datasources, complete the following steps. If you need help, refer to the WebSphere Application Server Advanced Edition documentation in the IBM WebSphere InfoCenter at: `http://www.ibm.com/software/webservers/appserv/doc/v40/aee/index.html`

1. Start the WebSphere Administrative Server by clicking **Start-> Programs-> IBM WebSphere-> Application Server V4.0 AE-> Start Admin Server**.
2. Start the WebSphere Administrative Console by clicking **Start-> Programs-> IBM WebSphere-> Application Server V4.0 AE-> Administrator's Console**.
3. Create a new JDBC provider entry (JDBC driver) with the following parameters:
   - Name: `OrderEntryJDBC`
   - Implementation class: `COM.ibm.db2.jdbc.DB2XADataSource`

– Driver file: <DB2_INSTALL_DIR>\SQLLIB\java\db2java.zip, where *<DB2_INSTALL_DIR>* is the DB2 installation directory.

> **Note:** Make sure you updated DB2 to use Java 2.

4. Create a new datasource with the following parameters:
   – JDBC provider: OrderEntryJDBC
   – Name: OrderEntryDS
   – JNDI name: jdbc/MyApp
   – Database name: ORDERENT
   – User: A user ID with sufficient authority to access the database
   – Password: A password for a user ID specified in the previous step
5. Create another datasource with the following parameters:
   – JDBC provider: OrderEntryJDBC or Sample DB Driver (this JDBC provider is created when you install WebSphere Application Server Advanced Edition)
   – Name: BRBDS
   – JNDI name: jdbc/BRBDS
   – Database name: BUSRULES
   – User: A user ID with sufficient authority to access the BUSRULES database
   – Password: A password for a user ID specified in the previous step

You need also create a separate application server for the OrderEntry application. There is a reason to create a new server instead of using the existing one. If you created an application server before you installed WebSphere Application Server Enterprise Edition, you won't see WebSphere Enterprise Services listed under the Custom tab for that application server. You can add these services manually. However, creating a new application server is easier than adding WebSphere Enterprise Services manually.

Follow these steps to create a new application server:

1. Run the **Create Application Server** wizard.
2. Type the name of the server OrderEntryServer.
3. Continue the wizard, and click **Finish** in the last screen.
4. Select **OrderEntryServer** in the Administrative Console.
5. Click the **Custom** tab.

You should see three WebSphere Enterprise Services as shown in Figure A-2.

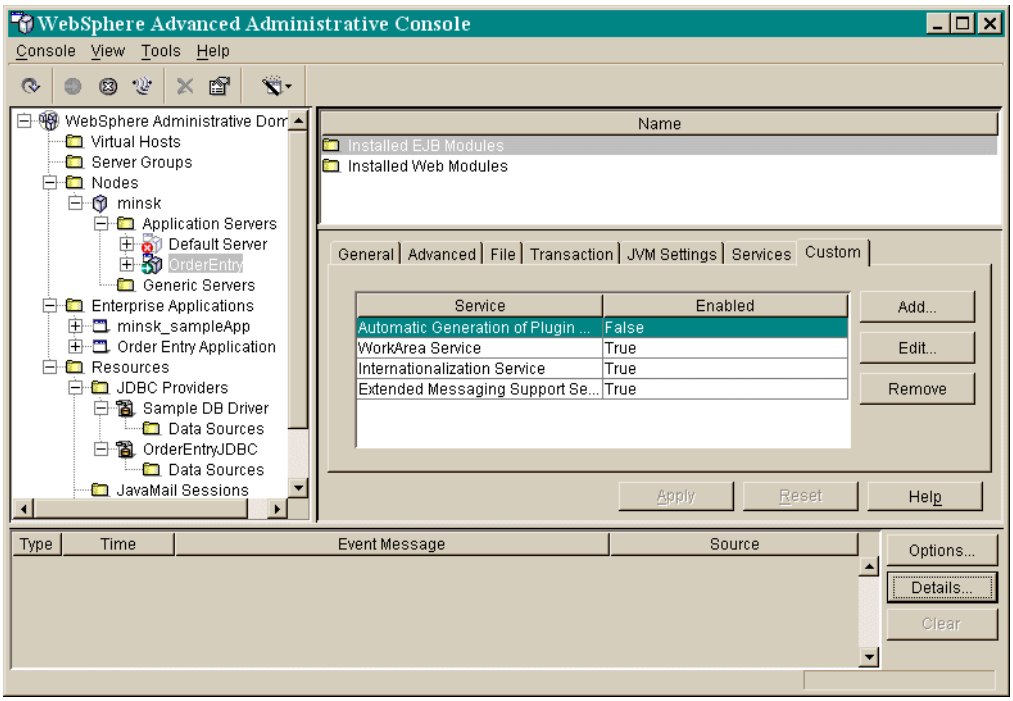


Figure A-2 Verifying the application server

6. If you plan to use the WebSphere Application Server internal HTTP server, you need to add an HTTP port information to the default_host virtual server.

> **Note:** Port information can be found by performing the following steps:
>
> 1. Select the **Services** tab on the OrderEntryServer configuration pane.
> 2. Select **Web Container Service** and click **Edit Properties**. Click the **Transport** tab to locate the HTTP port number.

Installing the OrderEntry application

The OrderEntry application relies on some platform/installation dependant data. This data is used by the application through the object of type `java.util.ResourceBundle`. This `ResourceBundle` object is created at runtime, based on two files supplied with the OrderEntry EAR file. You have to modify these two files according to your environment. Both of them have the same name, `OrderResourceBundle.properties`, but are located in two different JAR files within `OrderEntry.ear`: `Deployed_OrderEntryEJB.jar` and `OrderEntryWeb.war` (use WinZip or a similar utility to open the EAR and JAR files). The listing of the file contained within `Deployed_OrderEntryEJB.jar` is shown here:

```
factory=com.ibm.websphere.naming.WsnInitialContextFactory
system=localhost
port=900
schemaName=A_USER.
nameservice.ior.directory=<your ior directory path>
```

You have to change:

- `system` if you use a remote host
- `nameservice.ior.directory` if you plan to run the test using Orbix2000

The second file is almost identical; it just doesn't have the last line.

You can modify these files after you install the OrderEntry application and before you run it. One file is located in the %WAS_HOME%\installedApps\Order_Entry_Application.ear\OrderEntryWeb.war\ directory (see Figure A-3). Another file is located within Deployed_OrderEntryEJB.jar, which you can find in the %WAS_HOME%\installedApps\Order_Entry_Application.ear\ directory.

Important: If you plan to run the application on the local machine and don't want to install Orbix2000, you don't have to modify these files.

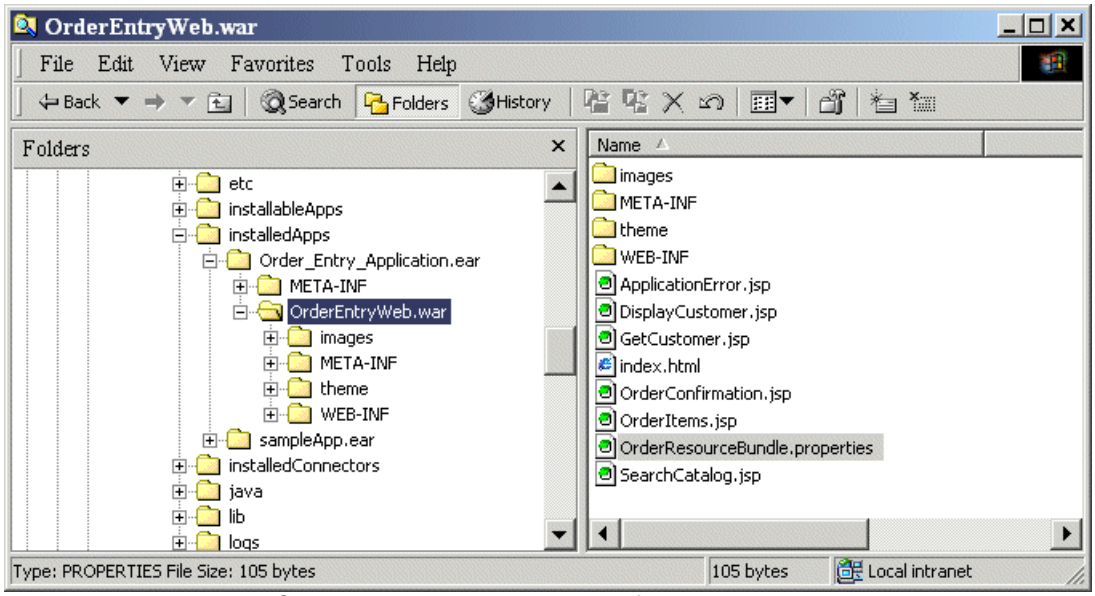


Figure A-3 Locating the OrderResourceBundle.properties file

Follow these steps to install the OrderEntry application:

Note: If you plan to go through the BRBeans development process, you don't have to install the OrderEntry application at this time. Skip the rest of this appendix.

1. Start the Install Enterprise Application wizard from the Administrative Console.
2. In the pop-up window, make sure you select the **Install Application (*.ear)** option.
3. Navigate to the **%OE_HOME%\OrderEntry.ear** file and select it.
4. Go through the wizard until you reach the Selecting Application Servers display.
5. Highlight all entries and click **Select Server**.
6. Select **OrderEntryServer**, and continue with the wizard.
7. When the WebSphere Application Server wizard asks if you want to re-generate the deployment code, select **No**.

Importing the BRBeans rules

Important: If you plan to go through the BRBeans development process, skip the rest of this appendix. The steps necessary to develop, deploy, and test the OrderEntry application with BRBeans are described in Chapter 3, “The Business Rule Beans service” on page 15.

The application is almost ready to be tested. Only the actual business rules that need to be used is missing. Follow these steps to import the business rules:

1. Start OrderEntryServer in WebSphere Administrative Console. Make sure there is no error in the Event Message pane of the Administrative Console.
2. Open a command prompt.
3. Run the following command:

   ```
   %OE_HOME%\BRBeans\ImportRules
   ```

   Note: The assumption is that you have installed WebSphere Enterprise Services in the %WAS_HOME%\Enterprise directory. If this is not true, you should edit the %OE_HOME%\BRBeans\ImportRules.bat file. Change the `%WAS_HOME%\Enterprise` line in the file to the directory name where you installed WebSphere Enterprise Services.

4. You should see no errors.
5. To verify that the rules were imported, run another command:

   ```
   %OE_HOME%\BRBeans\CheckRules
   ```

   Note: The assumption is that you installed WebSphere Enterprise Services in the %WAS_HOME%\Enterprise directory. If this is not true, you should edit the %OE_HOME%\BRBeans\CheckRules.bat file. Change the `%WAS_HOME%\Enterprise` line in the file to the directory name where you installed WebSphere Enterprise Services.

6. The rule management application is started. You should see the same folders as shown in Figure A-4.

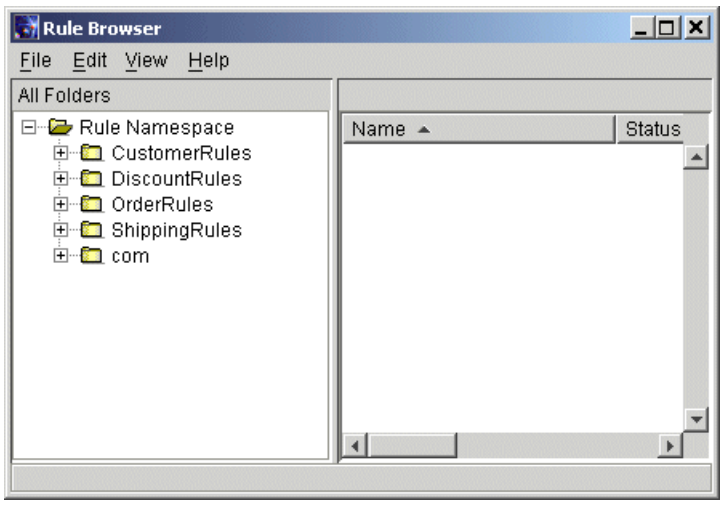


Figure A-4 Imported business rules

7. Close the rule management application.

Running the OrderEntry application

Now you're ready to test the application. To do that, perform the following steps:

1. Make sure the DB2 services are started.
2. Start the OrderEntry Server application server.
3. Start Internet Explorer (Netscape doesn't show pages correctly).
4. Point your browser to:

   `http://localhost:<port_for_OrderEntryServer>/OrderEntry/index.html`

   Here, *<port_for_OrderEntryServer>* is the port number that you used in step 6. on page 226.
5. In the browser window, click **Start the OrderEntry application**.
6. The application represents the online ordering system.
   a. Specify a customer ID in the range between 0001 and 0015.
   b. On the Select the Items Web page, specify an item number in the range 000001 to 000100 and the quantity.
   c. The alternative method is to click **Search Catalog** link, and select the items from the catalog. When you are done, click **Check Out** to record you order.

ActiveX application client-specific instructions

If you plan to run the ActiveX application client on your workstation, you need to perform the following configuration steps:

1. Install WebSphere Application Server Client on your client system (see "Installing the Advanced Edition client base" in IBM WebSphere InfoCenter).
2. Install the ActiveX Client support (see "Installing ActiveX client support (Windows only)" in the IBM WebSphere InfoCenter).
3. Place the `%OE_HOME%\ActiveX\OrderEntry.ear` file in the `%WAS_HOME%\WSsamples\Client\J2EE\` directory.

   Here *%WAS_HOME%* is where you installed the WebSphere Application Client, for example C:\IBM\WebSphere\AppClient.

   The EAR file is a slimmed down version of the full %OE_HOME%\OrderEntry.ear file. It only contains components necessary for J2EE client access.
4. Use the following command to launch the ActiveX client application:

   `%WAS_HOME%\Enterprise\bin\launchClientXJB %OE_HOME%\ActiveX\OrderEntryClient.exe`

   Put the path in "" if it contains spaces.
5. Fill out the bootstrap server and port fields in the options panel.

B

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG246504

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246504.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

File name	*Description*
OE.zip	All sample code in the ZIP format

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	2 GB minimum
Operating System:	Windows NT/2000
Processor:	450 MHz or higher
Memory:	512 MB

© Copyright IBM Corp. 2002

How to use the Web material

Refer to Appendix A, “Downloading and installing the OrderEntry application” on page 221, for installation instructions.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 234.

- *WebSphere Version 4 Application Development Handbook,* SG24-6134
- *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server,* SG24-5754

Other resources

MQSeries Using Java, SC34-5456, is also relevant as a further information source.

Referenced Web sites

These Web sites are also relevant as further information sources:

- WebSphere Studio Application Developer

  http://www.ibm.com/software/ad/studioappdev
- IBM WebSphere InfoCenter

  http://www.ibm.com/software/webservers/appserv/doc/v40/aee/index.html
- MQSeries classes for Java and MQSeries classes for Java Message Service

  http://www.ibm.com/software/ts/mqseries/txppacs/ma88.html
- MQSeries - Publish/Subscribe

  http://www.ibm.com/software/ts/mqseries/txppacs/ma0c.html
- Using Sybase PowerBuilder with the WebSphere Application Server, Enterprise Edition, Version 4 ActiveX to EJB Bridge

  http://www7b.boulder.ibm.com/wsdd/library/samples/powerbuilder.html
- The Object Management Group (OMG) Web site

  http://www.omg.org
- VisiBroker for C++ 4.5, installation guide

  http://www.borland.com/techpubs/books/vbcpp/vbcpp45/framesetindex.html
- VisiBroker 4.x for C++ download Web page

  http://www.borland.com/visibroker/download/vbcpp4x_steps.html
- Orbix 2000 2.0 Documentation Web page

  http://www.iona.com/docs/orbix2000/2.0/index.html
- Orbix E2A Enterprise Edition evaluation form

  http://www.iona.com/forms/asp_enterprise_eval_form.htm

© Copyright IBM Corp. 2002

- IONA Orbix 2000 Administrator's Guide

  http://www.iona.com/docs/orbix2000/2.0/admin_cpp/html/index.html
- IONA Orbix 2000 Programmer's Guide Java version

  http://www.iona.com/docs/orbix2000/2.0/pguide_java/html/index.html
- IONA Orbix 2000 Code Generation Toolkit Programmer's Guide

  http://www.iona.com/docs/orbix2000/2.0/codegen/html/index.html
- IONA Orbix 2000 Programmer's Guide Java version

  http://www.iona.com/docs/orbix2000/2.0/pguide_cpp/html/index.html
- Ant: the Jakarta project Web site

  http://jakarta.apache.org/ant/index.html
- Ant user manual

  http://jakarta.apache.org/ant/manual/index.html
- Java2 Platform Enterprise Edition Web site

  http://java.sun.com/j2ee
- JavaTM 2 SDK, Standard Edition Version 1.3.1_02 for Microsoft Windows

  http://java.sun.com/j2se/1.3/download-windows.html

How to get IBM Redbooks

Search for additional Redbooks or Redpieces, view, download, or order hardcopy from the Redbooks Web site:

ibm.com/redbooks

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become Redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Index

Symbols

<Listener> XML tag 66

A

ABC (Application Builder for Components) 210
ACID transaction 210
Active Server Pages 122
ActiveX 121
ActiveX bridge 3
  “dummy” client JAR file 149
  accessing Java objects, methods and fields 133
  client container approach 129
  client interface 126
  components of the client side EAR file 131
  initializing the JVM 128
  launching client container in an ActiveX process 131
  launching standard Java J2EE application in client container 131
  modifying EAR file for ActiveX J2EE client container 148
  no client container approach 129
  optimized container approach 131
  optimizing the EAR file 151
  using J2EE client container 129
ActiveX client application
  components 127
  installing 126
ActiveX client program design 124
ActiveX program
  accessing a Java class 136
  accessing Java objects 136
  Active Server Page guidelines 145
  case sensitivity 137
  client container guidelines 147
  error handling 142
  good programming guidelines 144
  handling arrays 140
  invoking Java methods 137
  passing “Object” type parameters 138
  primitive data type conversions 133
  return value 139
  thread 141
  Visual Basic guidelines 144
ActiveX_Config 127
ActiveX_EJBHomeRef 127
ActiveX_XJBHelpers 127
Activity context 215
activity wrapper process 214
Ant 191
Application Assembly Tool (AAT) 131, 149
Application Builder for Components (ABC) 210
arrays 140
automation container 140

B

base rule 18, 32, 33, 38
bean-managed transaction (BMT) 85
Bind 172, 185
BMT (bean-managed transaction) 85
bootstrapping 182
BPBeans 4, 209
  activity 212
  Activity context 215
  activity wrapper process 214
  Application Builder for Components (ABC) 210
  basics 210
  compensation mechanism 210
  compensation pair process 214
  concurrent process 214
  context 215
  development 216
  extended business transactions 210
  framework 210
  InOut Port 213
  Input Port 213
  instance cluster process 214
  JavaBean context 215
  keep alive process 214
  loosely coupled connection 211
  message cluster process 214
  outcome 212
  Outcome decider 215
  Outcome decider context 215
  Outcome Mapper 215
  OutIn Port 213
  Output Port 213
  port 213
  process 212, 213
  Process context 215
  programming model 212
  repeat process 214
  resource 214
  Resource context 215
  retry process 214
  sequential process 214
  source cluster process 214
  swap process 214
  task 212
  the LocalClientContext object 219
  time limit process 214
  Unit of Work (UOW) Director 214
BPBeans (Business Process Beans) 3
BRBeans 16
  base rules 18
  benefits 16
  classification rules 18
  combining strategy 35, 37
  components 16
  configuring business rules 23

© Copyright IBM Corp. 2002

creating rule implementors 23
deployment 47
EJBs 16, 17
filtering strategy 35
finding strategy 35
framework components 16
general approach for implementing business rules 22
identifying business rules 18
implementing a business rule with dependent rules 41
implementing a classifier rule 26
implementing base rule 32
implementing business rules 18
overview 16
performance considerations 46
Quick Copy 23
Rule Importer and Exporter 17
Rule Management APIs 17
rule states 25
setting up the development environment 19
strategies 32, 37
strategy objects 24
TriggerPoint 17
BRBeans (Business Rules Beans) 3
BRBeans EJBs 16
BRBeans performance
caching 46
database indexes 47
firing location 47
BRBeans rule 16, 25
firing location 26
BRBeans Rule Implementors 16
BRBeans strategies
combining strategy 25
filtering strategy 25
finding strategy 24
firing strategy 25
BRBeans Trigger Point 16
business logic 16
Business Process Beans (BPBeans) 3
Business Process Beans service 209
business process management 4
business rule 16, 18, 23, 39
dependent rule 41
Business Rules Beans (BRBeans) 3
Byte helper function 135

C

C++ CORBA client
configuring the environment 163
problem determination 185
VisiBroker client 177
C++ CORBA SDK 162
C++ CORBA server 187
configuring Orbix 188
configuring the environment 188
connecting to an Orbix server 190
problem determination 206
C++ CORBA software development kit (SDK) 162
caching 46
caller context 108
caller internationalization 109
accessing 113
case sensitivity 137
chg command 58
class module 127
classification rule 18
classified rule 38
classifier rule 26, 38
client 107
client application components 127
client container 122, 147, 148
client container approach 129
client interface 126
Client-side Internationalization (CSI) policy 108, 111
CMT (container-managed transaction) 85
coexistence 160
COM (Component Object Model) 121
combining strategy 25, 35, 37
Common Object Request Broker Architecture (CORBA) 154
compensation pair process 214
Component Object Model (COM) 121
concurrent process 214
container-managed transaction (CMT) 85
Controller 124
CORBA 102
building the test client 200
C++ CORBA clients for the OrderEntry application 163
connecting to nameservice 197
create and activate servant object 197
development 156
development for EJBs 158
EJB as CORBA client 161
EJB as CORBA server 161
enable CORBA traces 186
exporting object reference 197
IBM client of the Customer EJB 164
interoperability with an EJB 161
locating a server object 155
putting the ORB into an active state 198
role in J2EE 157
task 157
VisiBroker client of the Customer EJB 177
WebSphere Application Server Enterprise Edition CORBA support 159
WebSphere Application Server to third-party coexistence 160
WebSphere Application Server to third-party CORBA interoperability 159
WebSphere Application Server to WebSphere Application Server scenario 159
CORBA (Common Object Request Broker Architecture) 154
CORBA C++ Software Development Kit (SDK) 3
CORBA interoperability 3
CORBA Language Mapping 155
CORBA services 153, 155
CORBA support

WebSphere Application Server support scenarios 159
COrderClerk 128
CSI (Client-side Internationalization) 108
Currency helper function 135
Customer Table Layout (CSTMR) 11

D

database index 47
Def command 58
Delphi 123
dependent rule 41, 42, 44, 46
District Table Layout (Dstrct) 10

E

EJB
CORBA development 158
CORBA interoperability 161
creation wizard 68
Enterprise Java Bean v.2.0 52
Enterprise-class functionality 3
enterprise-class functions 2
enterprise-class quality of service (QoS) 2, 3
error handling 142
exception 142
exception handling 86
expired rule state 25
Extended Messaging Service 3, 51
extended messaging support 52

F

filtering strategy 25, 35
finding strategy 24, 35
firing location 26, 47
firing strategy 25
fixed mode 97

G

genie 191, 199
globalization 4

H

handling arrays 140

I

IBM Encina 2
IBM MQSeries 2
IBM TXSeries 2
IDL 166, 178, 191, 200, 202
IDL compiler 157, 162, 201
idlgen 200
IIOP (Internet-Inter ORB Protocol) 154
in effect rule state 25
instance cluster process 214
integration and adaptability 4
Interface Definition Language (IDL) 155
internationalization
caller 109
client-side 111
interface 109
invocation 110
server-side 111
internationalization context 107
caller context 108
Client-side Internationalization (CSI) policy 108
invocation context 108
management 111
Server-side Internationalization (SSI) 108
time zone 107
types of context 107
Internationalization context management 111
Internationalization service 3, 105
binding to 112
configuring the programming environment 116
enabling for application server 114
enabling internationalization context within EJB Java application client 116
Internationalization interface 109
InvocationInternationalization interface 110
overview 106
retrieving caller internationalization context 113
retrieving the invocation internationalization context 114
setting invocation locale and time zone 117
solution 106
tracing Internationalization service function 117
UserInternationalization interface 108
using in the example application 112
Internationalization Service API 108
Internet-Inter ORB Protocol (IIOP) 154
interoperable object reference (IOR) 156
interoperation 159
invalid rule state 25
invocation context 108
invocation internationalization 110
invocation locale and time zone 117
InvocationInternationalization interface 110
invoking component 107
IOR (interoperable object reference) 156, 182
Item Table Layout (ITEM) 13

J

J2EE
client container 129
role of CORBA 157
specification 157
J2EE Connector Architecture 4
Java 2 Platform, Enterprise Edition (J2EE) 2
Java Abstract Windowing Toolkit (AWT) 93
Java Message Service (JMS) Listener 3
Java methods 137
JavaBean context 215
JavaIDL 157
JClassFactory 128
JClassProxy 133
JMethodArgs 138

JMS
Admin tool 55, 78
advanced facilities 84
APIs 52
defining physical queues 55
defining resources 55
destination 52, 58
publish/subscribe model 76
support 52
testing sample code 74
JMS Listener 51, 52, 63, 80
bean-managed transaction (BMT) 85
begin() method 94
complete() method 96
container-managed transaction (CMT) 85
exception handling 86
JNDI entries 61
manager 63
MaxSessions 86
security 88
thread 86
threading 86
transaction support 84
wsqcf 85
wstcf 85
XML configuration file 65, 81
JMS point-to-point 51, 52
configuring 53
core components 54
JMS Provider 54, 61, 80
JMS publish/subscribe model 51, 53
clustering restriction 88
core components 77
message bean 82
WebSphere Application Server configuration 80
JNDI context definition 58
JNDI entry 61, 80
JNI 122
JObjectProxy 133

K

keep alive process 214
key-value-mode triplet 92

L

launchClientXJB 126
listener stanza 66
locale 107
locating a server object 155
loosely coupled connection 211

M

Main class 148
maximum send and receive size 103
MaxRetries 86
MaxSessions 86
MDB (message-driven bean) 52, 67, 89
message bean 52, 67, 82
developing 68
onMessage() 67
message broker 77
message cluster process 214
Message EJB 67
message-driven bean (MDB) 52, 67, 89
Microsoft Visual C++ 6.0 163
Model 124
Model/View/Controller (MVC) pattern 124
modMain 127
MQSeries 2
MQSeries clustering 88
MQSeries Explorer 55
MQSeries message broker 76, 77

N

namespace 185, 206
Naming Service 155
nested WorkArea 92, 97, 99
no client container approach 129
non-transactional queue connection factory 59
normal mode 97

O

Object Adapter 156
Object Request Broker (ORB) 154
Object type parameter 138
optimized container approach 131
ORB 171, 183, 196, 204
ORB (Object Request Broker) 154
Orbix 188, 190, 206
genie 191
Order Line Table Layout (ORDLIN) 12
Order Table Layout (ORDERS) 12
OrderEntry application 6
ABC Company 6
application flow 7
configuring WebSphere Application Server 224
customer transaction 7
database layout 10
description of the directory structure 223
development environment 14
downloading and installing 221
environment 222
importing the BRBeans rules 228
installing 226
introduction 6
running the OrderEntry application 229
setting up databases 224
tables 10
Outcome decider context 215
outcome set 212

P

physical queue 55
POA 195
POA (portable object adapter) 196
POA_Manager 195

point-to-point messaging model 51
point-to-point model 52
polymorphism 138
portable object adapter (POA) 196
PowerBuilder 123
primitive data type conversion 133
Process context 215
property 92
publish/subscribe messaging model 51
publish/subscribe model 53, 76

Q

queue connection factory 59
Quick Copy 23

R

read only mode 97
Redbooks Web site 234
  Contact us xi
Reflection API 139
repeat process 214
Resource context 215
retry process 214
return value 139
RMI-IIOP 158
root context 171, 184, 197, 205
rule client 16
rule client implementation 35
rule implementor 16, 22, 23
rule implementor class 26
Rule Importer and Exporter 17
Rule Management APIs 17
Rule Management Application 16, 22, 23, 28, 32
  Quick Copy 23
rule state 25
RuleImplementor 16
  fire method 27
  init method 26
  interface 23

S

scheduled rule state 25
security 88
send and receive size 103
sequential process 214
serializable 158
servant 155, 197
servant object 156
server application component 107
server object 155
Server Session Thread Pool 86
server-centric placement 125
Server-side Internationalization (SSI) policy 108, 111
setupCmdLineXJB 126
Shared WorkArea 3
source cluster process 214
SSI (Server-side Internationalization) policy 108, 111
Stock Table Layout (Stock) 13
strategy object 24
stub 156, 167, 179, 201
swap process 214

T

TCL 191, 200
thread 141
time limit process 214
topic 53
topic connection factory 80
traces 186, 206
transaction support 84
transactional queue connection factory 59
trigger point 16, 22, 24
Trigger Point framework 38
trigger point strategy 37
TriggerPoint 16, 30
TXSeries 2

U

unavailable rule state 25
Unit of Work (UOW) Director 214
UserInternationalization interface 108, 109
UserWorkArea 101
UserWorkArea interface 101

V

valuetype 158, 174, 176
valuetype library 162
variant 140
VBScript 123
View 124
VisiBroker 163, 177, 186
Visual Basic 122, 123
Visual C++ 122

W

WebSphere Application Server 60, 80, 159
  Advanced Edition 2
  Enterprise Edition 2
  passing WorkArea through multiple servers 104
  WorkArea configuration 102
WebSphere Application Server Advanced Edition 2
WebSphere Application Server Enterprise Edition 2
WebSphere Enterprise Services 1, 2, 3
WebSphere messaging 89
WebSphere Studio Application Developer 14, 18, 68
WorkArea 92
  accessing 95
  binding to facility 93
  changing the mode 101
  completing 96
  configuring 102
  configuring in WebSphere Application Server 102
  CORBA considerations 102
  creating 94
  creating and populating 94
  key-value-mode triplet 92

nested 92, 97, 99, 100
passing WorkArea through multiple WebSphere Application Servers 104
property modes 97
putting other objects 102
scope 92
terminating 96
terminating nested WorkArea 100
the UserWorkArea interface 101
using 95
using in the example application 93
WorkArea service 91
wsqcf 85
wstcf 85

X

XML configuration file 60, 65, 81

WebSphere Application Server Enterprise Edition 4.0

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

WebSphere Application Server Enterprise Edition 4.0

A Programmer's Guide

Redbooks

The ultimate reference to WebSphere Enterprise Services

Programming examples of Enterprise Services usage

Tips and techniques from the developers

With the advent of Java 2 Platform, Enterprise Edition (J2EE) application servers, companies are beginning to adopt the programming model that offers new levels of portability, flexibility, reuse, and scalability. Complementing J2EE technology, the emerging set of Web services standards arms companies with even greater business flexibility to compete in a changing world.

The most sophisticated e-businesses, however, need to extend these industry standards to meet important enterprise requirements. In response to the needs of the most demanding customers, IBM delivers the new WebSphere Enterprise Services that are part of WebSphere Application Server Enterprise Edition Version 4.0.

This IBM Redbook discusses the new WebSphere Enterprise Services and shows the best practices for using the services in the applications. The target audience for this book is the I/T specialist and application developers. Some parts of the book, such as Business Rule Beans and Business Process Beans, will be of particular interest to business domain experts.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-6504-00 ISBN 0738423998